XII Jornadas sobre Programación y Lenguajes

Sistedes 2012

ACTAS

JISBD PROLE JCIS

Sistedes
Sociedad de Ingeniería de Software y Tecnologías de Desarrollo de Software

IN LUMINE SAPIENTIA · UNIVERSITAS ALMERIENSIS ·

Almería, 17 al 19 de Septiembre

Editores: María del Mar Gallardo | Mateu Villaret | Luis Iribarne

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): Actas de las "XII Jornadas sobre Programación y Lenguajes (PROLE'2012), Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

PROLE 2012

XII Jornadas sobre Programación y Lenguajes (PROLE)

Almería, 17 al 19 de Septiembre de 2012

Editores:
María del Mar Gallardo
Mateu Villaret
Luis Iribarne

Actas de las "*XII Jornadas sobre Programación y Lenguajes (PROLE)*"
Almería, 17 al 19 de Septiembre de 2012
Editores: María del Mar Gallardo, Mateu Villaret, Luis Iribarne
http://sistedes2012.ual.es
http://www.sistedes.es

ISBN: 978-84-15487-27-2
Depósito Legal: AL 673-2012

Prólogo

El presente documento contiene los trabajos aceptados por los Comités de Programa para su presentación en las XII Jornadas de Programación y Lenguajes (PROLE'12) y el IV Taller de Programación funcional (TPF'12). Este año, y como es tradicional, PROLE y TPF se celebran en el marco de las Jornadas Sistedes 2012, en las que de forma paralela tienen lugar también las XVII Jornadas de Ingeniería del Software y Bases de Datos (JISBD'12) y las VIII Jornadas de Ciencia e Ingeniería de Servicios (JCIS'12), todas ellas bajo el auspicio de la Asociación de Ingeniería del Software y Tecnologías de Desarrollo de Software (SISTEDES, http://www.sistedes.es).

A lo largo de estos 12 años, desde su primera edición en Almagro, pasando por El Escorial, Málaga, Gijón o Granada, PROLE ha consolidado su papel como punto de encuentro y discusión de los investigadores españoles que estudian distintos aspectos (teóricos, aplicados o de implementación) relacionados con la programación.

En esta ocasión, considerando el momento de crisis profunda por la que está pasando España, es quizás más importante mantener jornadas como PROLE y TPF. Por eso nos gustaría dar las gracias, en primer lugar, a los autores de los artículos enviados. Los trabajos que se presentan este año muestran las distintas líneas de investigación que actualmente se están desarrollando en nuestras Universidades e Institutos de Investigación. Podemos encontrar artículos que tratan aspectos fundamentales de los lenguajes (considerando distintos paradigmas), con un fuerte contenido formal, y artículos más aplicados en los que se da más relevancia a los aspectos de implementación. Podemos encontrar trabajos maduros, que han sido presentados o publicados en reuniones de carácter internacional, artículos completos inéditos, y trabajos en progreso, que están aún en una estado más preliminar. En cualquier caso, los trabajos presentados son, en general, de gran calidad técnica lo que me permite afirmar que, a pesar de las dificultades, la investigación en Programación en España goza de una salud excelente.

El programa se completa con la Conferencia titulada "Strategy-Driven Graph Transformations in PORGY" impartida por la Dra. Maribel Fernández del King's College London, a quien agradecemos su participación.

Por su parte, TPF'12 ha seleccionado tres trabajos directamente relacionados con el paradigma de programación funcional. El programa de TPF'12 cuenta este año además con dos charlas invitadas sobre la aplicación de la programación funcional en la industria, impartidas por Diana Corbacho y José Iborra, de un seminario sobre cálculo lambda y reducciones, a cargo de Pablo Nogueira, y de un tutorial de paralelismo en Haskell a cargo de Ricardo Peña. A todos ellos queremos agradecerles su buena predisposición e iniciativa.

Asimismo, nos gustaría dar las gracias a los Comités de Programa por su trabajo en la revisión y selección de los artículos, y a los revisores externos que han colaborado desinteresadamente.

Por último, no queremos dejar escapar la ocasión para recordar de manera especial al profesor Víctor Gulías, estrechamente vinculado a estas jornadas, y que ha fallecido recientemente a una temprana edad. Te echaremos de menos.

Para terminar, es justo mencionar la magnífica ciudad que nos acoge, Almería, y el grupo de investigación que ha organizado todo el evento. Nos gustaría agradecer al profesor Luis Iribarne, y al comité de organización en su conjunto, su paciencia y su trabajo durante este último año que, estamos seguros, culminará con el éxito de las Jornadas Sistedes 2012.

Septiembre 2012, Almería
María del Mar Gallardo
Mateu Villaret
Presidentes de los Comités de Programa de PROLE'12 y TPF'12.

Prologo de la Organización

Las jornadas SISTEDES 2012 son un evento científico-técnico nacional de ingeniería y tecnologías del software que se celebra este año en la Universidad de Almería durante los días 17, 18 y 19 de Septiembre de 2012, organizado por el Grupo de Investigación de Informática Aplicada (TIC-211). Las Jornadas SISTEDES 2012 están compuestas por las XVII Jornadas de Ingeniería del Software y de Bases de Datos (JISBD'2012), las XII Jornadas sobre Programación y Lenguajes (PROLE'2012), y la VIII Jornadas de Ciencia e Ingeniería de Servicios (JCIS'2012). Durante tres días, la Universidad de Almería alberga una de las reuniones científico-técnicas de informática más importantes de España, donde se exponen los trabajos de investigación más relevantes del panorama nacional en ingeniería y tecnología del software. Estos trabajos están auspiciados por importantes proyectos de investigación de Ciencia y Tecnología financiados por el Gobierno de España y Gobiernos Regionales, y por proyectos internacionales y proyectos I+D+i privados. Estos encuentros propician el intercambio de ideas entre investigadores procedentes de la universidad y de la empresa, permitiendo la difusión de las investigaciones más recientes en ingeniería y tecnología del software. Como en ediciones anteriores, estas jornadas están auspiciadas por la Asociación de Ingeniería del Software y Tecnologías de Desarrollo de Software (SISTEDES).

Agradecemos a nuestras entidades colaboradoras, Ministerio de Economía y Competitividad (MINECO), Junta de Andalucía, Diputación Provincial de Almería, Ayuntamiento de Almería, Vicerrectorado de Investigación, Vicerrectorado de Tecnologías de la Información (VTIC), Enseñanza Virtual (EVA), Escuela Superior de Ingeniería (ESI/EPS), Almerimatik, ICESA, Parque Científico-Tecnológico de Almería (PITA), IEEE España, Colegio de Ingenieros Informática de Andalucía, Fundación Mediterránea, y a la Universidad de Almería por el soporte facilitado. Asimismo a D. Félix Faura, Director de la Agencia Nacional de Evaluación y Prospectiva (ANEP) de la Secretaría de Estado de I+D+i, Ministerio de Economía y Competitividad, a D. Juan José Moreno, Catedrático de la Universidad Politécnica de Madrid, presidente de la Sociedad de Ingeniería y Tecnologías del Software (SISTEDES), a D. Francisco Ruiz, Catedrático de la Universidad de Castilla-La Mancha, y a D. Miguel Toro, Catedrático de la Universidad de Sevilla, por su participación en la mesa redonda "*La investigación científica informática en España y el año Turing*"; a Armando Fox de la Universidad de Berkley (EEUU) y a Maribel Fernández del King's College London (Reino Unido), como conferenciantes principales de las jornadas, y a los presidentes de las tres jornadas por facilitar la confección de un programa de *Actividades Turing*. Especial agradecimiento a los voluntarios de las jornadas SISTEDES 2012, estudiantes del Grado de Ingeniería Informática y del Postgrado de Doctorado de Informática de la Universidad de Almería, y a todo el equipo del Comité de Organización que han hecho posible con su trabajo la celebración de una nueva edición de las jornadas JISBD'2012, PROLE'2012 y JCIS'2012 (jornadas SISTEDES 2012) en la Universidad de Almería.

Luis Iribarne
Presidente del Comité de Organización
@sistedes2012{JISBD;PROLE;JCIS}

Comité Científico del PROLE'2012

Presidente del Comité:

María del Mar Gallardo, Universidad de Málaga

Miembros:

Jesús Almendros, Universidad de Almería
María Alpuente, Universidad Politécnica de Valencia
Puri Arenas, Universidad Complutense de Madrid
Miquel Bertrán, Universidad Ramón Llull
Rafael Caballero, Universidad Complutense de Madrid
Manuel Carro, Universidad Politécnica de Madrid
Fernando Cuartero, Universidad de Castilla la Mancha
Francisco Durán, Universidad de Málaga
Miguel Gómez-Zamalloa, Universidad Complutense de Madrid
Salvador Lucas, Universidad Politécnica de Valencia
Paqui Lucio, Universidad del País Vasco
Ginés Moreno, Universidad de Castilla la Mancha
Juan José Moreno, Universidad Politécnica de Madrid
Marisa Navarro, Universidad del País Vasco
Javier Oliver, Universidad Politécnica de Valencia
Albert Oliveras, Universidad Politécnica de Cataluña
Fernando Orejas, Universidad Politécnica de Cataluña
Ricardo Peña, Universidad Complutense de Madrid
César Sánchez, Fundación IMDEA Software
Alicia Villanueva, U. Politécnica de Valencia

Comité Científico de TPF'12

Presidente del Comité:

Mateu Villaret, Universidad de Girona

Miembros:

Miquel Bofill, Universidad de Girona
Laura Castro, Universidad de A Coruña
Emilio J. Gallego, Universidad Politécnica de Madrid
Francisco Gutiérrez, Universidad de Málaga
Raúl Gutiérrez, University of Illinois at Urbana-Champaign
Pablo Nogueira, Universidad Politécnica de Madrid
Ricardo Peña, Universidad Complutense de Madrid
Jaime Sánchez, Universidad Complutense de Madrid
Josep Silva, Universidad Politécnica de Valencia

Revisores Externos de PROLE'12 y TPF'12:

Javier Álvez
Javier Espert
Álvaro Fernández
Francisco Frechina
Lars-ke Fredlund
Álvaro García
Montserrat Hermo
Fernando López
Víctor Pablos-Ceruelo
Fernando Pérez
Rafael del Vado
Carlos Vázquez

Comité de Organización

Presidente:

Luis Iribarne (Universidad de Almería)

Miembros:

Alfonso Bosch (Universidad de Almería)
Antonio Corral (Universidad de Almería)
Diego Rodríguez (Universidad de Almería)
Elisa Álvarez, Fundación Mediterránea
Javier Criado (Universidad de Almería)
Jesús Almendros (Universidad de Almería)
Jesús Vallecillos (Universidad de Almería)
Joaquín Alonso (Universidad de Almería)
José Andrés Asensio (Universidad de Almería)
José Antonio Piedra (Universidad de Almería)
José Francisco Sobrino (Universidad de Almería)
Juan Francisco Inglés (Universidad Politécnica de Cartagena)
Nicolás Padilla (Universidad de Almería)
Rosa Ayala (Universidad de Almería)
Saturnino Leguizamón (Universidad Tecnológica Nacional, Argentina)

Índice de Contenidos del PROLE'2012

Charla Invitada

Maribel Fernández. *Strategy-Driven Graph Transformations in PORGY* 297

Sesión 1: Programación (lógica) funcional/Aplicaciones
Chair: Dr. Ricardo Peña

Lidia Sánchez Gil, Mercedes Hidalgo-Herrero and Yolanda Ortega- Mallén. *A formalization of Launchbury's natural semantics for lazy evaluation in Coq* 15-29

Ignacio Castiñeiras and Fernando Saenz-Perez. *Improving the Performance of FD Constraint Solving in a CFLP System* 31-32

David Duque and Laura M. Castro. *Tecnología funcional en aplicaciones de televisión interactiva: acceso a redes sociales con Synthetrick* 33-47

Sesión 2: **Transformación de programas**
Chair: Dr. Jesús Almendros

Jose F. Morales, Rémy Haemmerlé, Manuel Carro and Manuel Hermenegildo. *Lightweight Compilation of (C)LP to JavaScript* 51-52

David Insa, Josep Silva and César Tomás. *Mejora del rendimiento de la depuración declarativa mediante compresión y expansión de bucles* 53-67

Alejandro Acosta, Francisco Almeida and Vicente Blanco. *Paralldroid: a source to source translator for development of native Android applications* 69-83

Sesión 3: Bases de Datos
Chair: Dr. Ginés Moreno

Fernando Saenz-Perez. *Tabling with Support for Relational Features in a Deductive Database* 87-101

Rafael Caballero, Jose Luzon-Martin and Antonio Tenorio. *Test-Case Generation for SQL Nested Queries with Existential Conditions* 103-117

Jesús M. Almendros-Jiménez, Alejandro Luna Tedesqui and Ginés Moreno. *Debugging Fuzzy-XPath Queries* 119-133

Sesión 4: Programación lógica/restricciones
Chair: Dr. Fernando Orejas

Pablo Chico De Guzmán, Manuel Carro, Manuel Hermenegildo and Peter Stuckey. *A General Implementation Framework for Tabled CLP* 137-138

Marco Comini, Laura Titolo and Alicia Villanueva. *Abstract diagnosis for timed concurrent constraint programs* 139-140

Miquel Bofill Arasa, Joan Espasa Arxer, Miquel Palahí Sitges and Mateu Villaret. *An extension to Simply for solving Weighted Constraint Satisfaction Problems with Pseudo-Boolean Constraints* 141-155

Sesion 5: Fundamentos
Chair: Dra. Paqui Lucio

Edelmira Pasarella, Fernando Orejas, Elvira Pino and Marisa Navarro. *Semantics of structured normal logic programs* 159-160

Simone Santini. *Regular queries in event systems with bounded uncertainty*..... 161-175

Pedro J. Morcillo, Gines Moreno, Jaime Penabad and Carlos Vázquez. *String-based Multi-adjoint Lattices for Tracing Fuzzy Logic Computations* 177-191

Sesion 6: Ingeniería del Software
Chair: Dra. Marisa Navarro

Dragan Ivanovic, Manuel Carro and Manuel Hermenegildo. *Constraint-Based Runtime Prediction of SLA Violations in Service Orchestrations* 195-196

Jesus M. Almendros-Jimenez and Luis Iribarne. *PTL: A Prolog based Model Transformation Language*..... 197-211

Enrique Chavarriaga, Fernando Díez and Alfonso Díez. *Intérprete PsiXML para gramáticas de mini-Lenguajes XML en aplicaciones Web* 213-227

Sesion 7: Lógica Temporal/Model checking
Chair: Dra. Alicia Villanueva

Jose Gaintzarain, Montserrat Hermo, Paqui Lucio, Marisa Navarro and Fernando Orejas. *Invariant-Free Clausal Temporal Resolution*..... 231-232

Damián Adalid, Alberto Salmerón, María del Mar Gallardo and Pedro Merino. *Testing Temporal Logic on Infinite Java Traces* 233-234

Laura Panizo and María del Mar Gallardo. *Analyzing Hybrid Systems with JPF*..... 235-249

Índice de Contenidos del TPF'2012

Sesión 1: Ponencia invitada y seminario

Jose Iborra. *Eden: An F#/WPF framework for building GUI tools* 253-254

Pablo Nogueira. *Los cálculos lambda y lambda-value y sus estrategias de reducción*.....259

Sesión 2: Artículos

Alvaro Garcia-Perez and Pablo Nogueira. *Enfoque Normal y Enfoque Spine para Reducción en el Cálculo Lambda Puro* 261

David Insa, Josep Silva, Salvador Tamarit and César Tomás. *Fragmentación de Programas con Emparejamiento de Patrones* 265-266

Macías López Iglesias, Laura M. Castro and David Cabrero. *Sistema funcional distribuido de publicidad para iDTV* 267-291

Sesión 3: Tutorial y ponencia invitada

Ricardo Peña. *Programación con paralelismo de datos en Haskell* 277-291

Diana Parra. *Erlang, del laboratorio a la empresa* 295

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): Actas de las "XII Jornadas sobre Programación y Lenguajes (PROLE'2012), Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

Sesión 1

Programación (lógica) Funcional/Aplicaciones

Chair: *Dr. Ricardo Peña*

Sesión 1: Programación (lógica) funcional/Aplicaciones
Chair: Dr. Ricardo Peña

Lidia Sánchez Gil, Mercedes Hidalgo-Herrero and Yolanda Ortega- Mallén. *A formalization of Launchbury's natural semantics for lazy evaluation in Coq*.

Ignacio Castiñeiras and Fernando Saenz-Perez. *Improving the Performance of FD Constraint Solving in a CFLP System*.

David Duque and Laura M. Castro. *Tecnología funcional en aplicaciones de televisión interactiva: acceso a redes sociales con Synthetrick*.

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): Actas de las "XII Jornadas sobre Programación y Lenguajes (PROLE'2012), Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

A formalization in Coq of Launchbury's natural semantics for lazy evaluation*

Lidia Sánchez-Gil[1] **Mercedes Hidalgo-Herrero**[2] **Yolanda Ortega-Mallén**[3]

[1,3]Dpt. Sistemas Informáticos y Computación, Facultad de CC. Matemáticas, Universidad Complutense de Madrid, Spain, lidiasg@mat.ucm.es, yolanda@sip.ucm.es
[2]Dpt. Didáctica de las Matemáticas, Facultad de Educación, Universidad Complutense de Madrid, Spain, mhidalgo@edu.ucm.es

Abstract: We are working on the implementation of Launchbury's semantics for lazy evaluation in the proof assistant Coq. We use a locally nameless representation where names are reserved for free variables, while bound variable names are replaced by indices. This avoids the need of α-conversion and Barendregt's variable convention, and facilitates the formalization in Coq. Simultaneous recursive local declarations in the calculus require the management of multibinders and the use of mutually inductive types.

Keywords: Formalization, locally nameless representation, proof assistant, Coq, natural semantics, lazy evaluation.

1 Motivation

Call-by-need evaluation, which avoids repeated computations, is the semantic foundation for lazy functional programming languages like Haskell or Clean. Launchbury defines in [Lau93] a natural semantics for lazy evaluation where the set of *bindings*, i.e., (variable, expression) pairs, is explicitly handled to make possible their sharing. To prove that this lazy semantics is *correct* and *computationally adequate* with respect to a standard denotational semantics, Launchbury defines an alternative semantics. On the one hand, functional application is modeled denotationally by extending the environment with a variable bound to a value. This new variable represents the formal parameter of the function, while the value corresponds to the actual argument. For a closer approach of this mechanism, applications are carried out in the alternative semantics by introducing indirections instead of by performing the β-reduction through substitution. On the other hand, the update of bindings with their computed values is an operational notion without a denotational counterpart. Thus, the alternative semantics does no longer update bindings and becomes a *call-by-name* semantics.

Alas, the proof of the equivalence between the lazy semantics and its alternative version is detailed nowhere, and a simple induction turns out to be insufficient. Intuitively, both reduction systems should produce the same results, but this cannot be directly established. Values may contain free variables that depend on the context of evaluation, which is represented by the heap of bindings. The changes introduced by the alternative semantics do deeply affect these heaps.

* Work partially supported by the projects TIN2009-14599-C03-01 and S2009/TIC-1465.

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): PROLE'2012, pp. 15-29, ISBN:978-84-15487-27-2.
Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

Although indirections and "duplicated" bindings (a consequence of not updating) do not add relevant information to the context, it is awkward to prove this fact.

We intend to prove formally the equivalence between Launchbury's semantics and its alternative version. In the usual representation of the λ-calculus, i.e., with variable names for free and bound variables, terms are identifed up to α-conversion. Dealing with α-equated terms usually implies the use of Barendregt's variable convention [Bar84] to avoid the renaming of bound variables. However, the use of the variable convention in rule inductions is sometimes dubious and may lead to *faulty* results (as it is shown by Urban et al. in [UBN07]). Looking for a system of binding more amenable to formalization, we have chosen a *locally nameless* representation (as presented by Charguéraud in [Cha11]). This is a mixed notation where bound variable names are replaced by de Bruijn indices [dB72], while free variables preserve their names. Hence, α-conversion is no longer needed and variable substitution is easily defined because there is no danger of name capture. Moreover, this representation is suitable for working with proof assistants like Coq [Ber06] or Isabelle [NPW02]. We have preferred Coq to Isabelle for our formalization because Coq allows the definition of inductive types, that are needed for the recursive local declarations, while we encountered some problems when we first tried with Isabelle. Furthermore, Coq offers dependent types and proof by reflection, that will be helpful for our theory and proof developments. Agda [AGD] allows dependent and inductive types too, but offers a lesser degree of internal automation and does not support tactical theorem proving.

Our concern for reproducing and formalizing the proof of this equivalence is not arbitrary. Launchbury's natural semantics has been cited frequently and has inspired many further works as well as several extensions of his semantics [BKT00, Ses97, EM07], where the corresponding adequacy proofs have been obtained by just adapting Launchbury's proof scheme. We have extended ourselves the λ-calculus with a new expression that introduces parallelism in functional applications [SHO10].

The paper is structured as follows: In Section 2 we present the locally nameless representation of the λ-calculus extended with mutually recursive local declarations. In Section 3, we express Launchbury's semantic rules in the new style and present several properties of the reduction system that are useful for the equivalence proof.[1] In Section 4 we describe the current state of the implementation in Coq of this locally nameless translation of the syntax and the semantics. In Section 5 we comment on some related work. In the last section we explain what we have achieved so far and what remains to be done.

2 The locally nameless representation

The language described by Launchbury in [Lau93] is a *normalized* λ-calculus extended with recursive local declarations. We reproduce the restricted syntax in Figure 1.a. Normalization is achieved in two steps. First an α-conversion is carried out so that all bound variables have distinct names. In a second phase, arguments for applications are enforced to be variables. These *static* transformations simplify the definition of the reduction rules.

Next, we follow the methodology summarized in [Cha11]:

[1] The "paper-and-pencil" proofs of these lemmas and other auxiliary results are detailed in [SHO12].

$$x \in Var \qquad e \in Exp ::= \lambda x.e \mid (e\ x) \mid x \mid \texttt{let}\ \{x_i = e_i\}_{i=1}^{n}\ \texttt{in}\ e.$$

(a) Restricted *named* syntax

$$x \in Id \qquad i, j \in \mathbb{N}$$
$$v \in Val \quad ::= \texttt{bvar}\ i\ j \mid \texttt{fvar}\ x$$
$$t \in LNExp ::= v \mid \texttt{abs}\ t \mid \texttt{app}\ t\ v \mid \texttt{let}\ \{t_i\}_{i=1}^{n}\ \texttt{in}\ t$$

(b) Locally nameless syntax

Figure 1: Extended λ-calculus

1. Define the syntax of the extended λ-calculus in the locally nameless style.
2. Define the variable opening and variable closing operations.
3. Define the free variables and substitution functions, as well as the local closure predicate.
4. State and prove the properties of the operations on terms that are needed in the development to be accomplished.

2.1 Locally nameless syntax

The locally nameless (restricted) syntax is shown in Figure 1.b. *Var* stands now for the set of *variables*, where we distinguish between *bound variables* and *free variables*. The calculus includes two binding constructions: λ-abstraction and `let`-declaration. Being the latter a *multi-binder*, we follow Charguéraud [Cha11] and represent bound variables with two natural numbers: The first number is a de Bruijn index that counts how many binders (abstraction or `let`) one needs to cross to the left to reach the corresponding binder for the variable, while the second refers to the position of the variable inside that binder. Abstractions are seen as multi-binders that bind only one variable; thus, the second number should be zero. In the following, we will represent a list like $\{t_i\}_{i=1}^{n}$ as $\overline{t}$, with length $|\overline{t}| = n$.

Example 1 *Let $e \in Exp$ be an expression in the named representation:*

$$e \equiv \lambda z.\texttt{let}\ x_1 = \lambda y_1.y_1, x_2 = \lambda y_2.y_2, x_3 = x\ \texttt{in}\ (z\ x_2).$$

The corresponding locally nameless term $t \in LNExp$ is:

$$t \equiv \texttt{abs}\ (\texttt{let}\ \texttt{abs}\ (\texttt{bvar}\ 0\ 0), \texttt{abs}\ (\texttt{bvar}\ 0\ 0), \texttt{fvar}\ x\ \texttt{in}\ \texttt{app}\ (\texttt{bvar}\ 1\ 0)\ (\texttt{bvar}\ 0\ 1)).$$

Notice that x_1 and x_2 denote α-equivalent expressions in e. This is more clearly seen in t, where both expressions are represented with syntactically equal terms.

As bound variables are nameless, the first phase of Launchbury's normalization is unneeded. However, application arguments are still restricted to variables.

2.2 Variable opening and variable closing

Variable opening and *variable closing* are the main operations to manipulate locally nameless terms. We extend the definitions given by Charguéraud in [Cha11] to the `let`-declaration.[2]

[2] Multiple binders are defined in [Cha11]. Two constructions are given: One for non-recursive local declarations, and another for mutually recursive expressions. Yet both extensions are not completely developed.

$$
\begin{array}{lll}
\{k \to \overline{x}\}(\texttt{bvar}\ i\ j) & = \begin{cases} \texttt{fvar}\ (\texttt{List.nth}\ j\ \overline{x}) & \text{if } i = k \wedge j < |\overline{x}| \\ \texttt{bvar}\ i\ j & \text{otherwise} \end{cases} \\
\{k \to \overline{x}\}(\texttt{fvar}\ x) & = \texttt{fvar}\ x \\
\{k \to \overline{x}\}(\texttt{abs}\ t) & = \texttt{abs}\ (\{k+1 \to \overline{x}\}\ t) \\
\{k \to \overline{x}\}(\texttt{app}\ t\ v) & = \texttt{app}\ (\{k \to \overline{x}\}\ t)\ (\{k \to \overline{x}\}\ v) \\
\{k \to \overline{x}\}(\texttt{let}\ \overline{t}\ \texttt{in}\ t) & = \texttt{let}\ (\{k+1 \to \overline{x}\}\ \overline{t})\ \texttt{in}\ (\{k+1 \to \overline{x}\}\ t)
\end{array}
$$

where $\{k \to \overline{x}\}\ \overline{t} = \texttt{List.map}\ (\{k \to \overline{x}\}\cdot)\ \overline{t}$.

Figure 2: Variable opening

To explore the body of a binder (abstraction or `let`), the corresponding bound variables are replaced with fresh names. In the case of `abs` t the *variable opening operation* replaces in t with a (fresh) name every bound variable that refers to the outermost abstraction. Similarly, to open `let` $\overline{t}$ `in` t we provide a list of $|\overline{t}|$ distinct fresh names to replace those bound variables that occur in $\overline{t}$ and in t which refer to this particular declaration.

Variable opening is defined by means of a more general function $\{k \to \overline{x}\}t$ (Figure 2), where the number k represents the nesting level of the binder to be opened, and $\overline{x}$ is a list of pairwise-distinct identifiers in Id. Since the level of the outermost binder is 0, variable opening is defined as $t^{\overline{x}} = \{0 \to \overline{x}\}t$. We extend this operation to lists of terms: $\overline{t}^{\overline{x}} = \texttt{List.map}\ (\cdot^{\overline{x}})\ \overline{t}$.

The last definition and those in Figure 2 include some operations on lists. We use an ML-like notation. For instance, `List.nth` $j\ \overline{x}$ represents the $(j+1)^{th}$ element of $\overline{x}$,[3] and `List.map` $f\ \overline{t}$ applies function f to every term in $\overline{t}$. In the rest of definitions we will use similar list operations.

Example 2 *Consider the term* `abs (let bvar 0 1, bvar 1 0 in app (abs bvar 2 0) (bvar 0 1))`. *Hence, the body of the abstraction is:*

$$u \equiv \texttt{let bvar 0 1,}\boxed{\texttt{bvar 1 0}}\texttt{ in app (abs }\boxed{\texttt{bvar 2 0}}\texttt{) (bvar 0 1)}.$$

But then in u the bound variables referring to the outermost abstraction (shown squared) point to nowhere. Therefore, we consider $u^{[x]}$ *instead of u, where*

$$
\begin{array}{lll}
u^{[x]} & = & \{0 \to x\}(\texttt{let bvar 0 1, bvar 1 0 in app (abs bvar 2 0) (bvar 0 1)}) \\
& = & \texttt{let}\ \{1 \to x\}(\texttt{bvar 0 1, bvar 1 0})\ \texttt{in}\ \{1 \to x\}(\texttt{app (abs bvar 2 0) (bvar 0 1)}) \\
& = & \texttt{let bvar 0 1, fvar } x\ \texttt{in app (abs }\{2 \to x\}(\texttt{bvar 2 0})\texttt{) (bvar 0 1)} \\
& = & \texttt{let bvar 0 1, fvar } x\ \texttt{in app (abs fvar } x\texttt{) (bvar 0 1)}
\end{array}
$$

Inversely to variable opening, there is an operation to transform free names into bound variables. The *variable closing* of t is represented by ${}^{\backslash\overline{x}}t$, where $\overline{x}$ is the list of names to be bound (recall that the names in $\overline{x}$ are distinct). Again, a general function $\{k \leftarrow \overline{x}\}t$ (Figure 3) is defined. Whenever `fvar` x is encountered, x is looked up in $\overline{x}$: If x occurs in position j, then the free variable is replaced by `bvar` $k\ j$, otherwise it is left unchanged. Variable closing is then defined as ${}^{\backslash\overline{x}}t = \{0 \leftarrow \overline{x}\}t$. Its extension to lists is ${}^{\backslash\overline{x}}\overline{t} = \texttt{List.map}\ ({}^{\backslash\overline{x}}\cdot)\ \overline{t}$.

Example 3 *We close the term obtained by opening the body of the abstraction in Example 2 (i.e., the term u):* $t \equiv$ `let bvar 0 1, fvar` x `in app (abs fvar` x`) (bvar 0 1)`.

[3] In order to better accommodate to bound variables indices, elements in a list are numbered starting with 0.

$$
\begin{array}{ll}
\{k \leftarrow \overline{x}\}(\texttt{bvar}\ i\ j) & = \texttt{bvar}\ i\ j \\
\{k \leftarrow \overline{x}\}(\texttt{fvar}\ x) & = \begin{cases} \texttt{bvar}\ k\ j & \text{if } \exists j : 0 \leq j < |\overline{x}|.x = \texttt{List.nth}\ j\ \overline{x} \\ \texttt{fvar}\ x & \text{otherwise} \end{cases} \\
\{k \leftarrow \overline{x}\}(\texttt{abs}\ t) & = \texttt{abs}\ (\{k+1 \leftarrow \overline{x}\}\ t) \\
\{k \leftarrow \overline{x}\}(\texttt{app}\ t\ v) & = \texttt{app}\ (\{k \leftarrow \overline{x}\}\ t)\ (\{k \leftarrow \overline{x}\}\ v) \\
\{k \leftarrow \overline{x}\}(\texttt{let}\ \overline{t}\ \texttt{in}\ t) & = \texttt{let}\ (\{k+1 \leftarrow \overline{x}\}\ \overline{t})\ \texttt{in}\ (\{k+1 \leftarrow \overline{x}\}\ t)
\end{array}
$$

where $\{k \leftarrow \overline{x}\}\ \overline{t} = \texttt{List.map}\ (\{k \leftarrow \overline{x}\}\ \cdot)\ \overline{t}$.

Figure 3: Variable closing

$$
\text{LC\_VAR}\ \frac{}{\texttt{lc}\ (\texttt{fvar}\ x)} \qquad \text{LC\_ABS}\ \frac{\forall x \notin L \subseteq Id \quad \texttt{lc}\ t^{[x]}}{\texttt{lc}\ (\texttt{abs}\ t)} \qquad \text{LC\_APP}\ \frac{\texttt{lc}\ t \quad \texttt{lc}\ v}{\texttt{lc}\ (\texttt{app}\ t\ v)}
$$

$$
\text{LC\_LET}\ \frac{\forall \overline{x}^{|\overline{t}|} \notin L \subseteq Id \quad \texttt{lc}\ [t : \overline{t}]^{\overline{x}}}{\texttt{lc}\ (\texttt{let}\ \overline{t}\ \texttt{in}\ t)} \qquad \text{LC\_LIST}\ \frac{\texttt{List.forall}\ (\texttt{lc}\ \cdot)\ \overline{t}}{\texttt{lc}\ \overline{t}}
$$

Figure 4: Local closure

$$
\begin{array}{rcl}
\backslash^{x} t & = & \{0 \leftarrow x\}(\texttt{let}\ \{\texttt{bvar}\ 0\ 1, \texttt{fvar}\ x\}\ \texttt{in}\ \texttt{app}\ (\texttt{abs}\ (\texttt{fvar}\ x))\ (\texttt{bvar}\ 0\ 1)) \\
& = & \texttt{let}\ \{1 \leftarrow x\}(\texttt{bvar}\ 0\ 1, \texttt{fvar}\ x)\ \texttt{in}\ \{1 \leftarrow x\}(\texttt{app}\ (\texttt{abs}\ \texttt{fvar}\ x)\ (\texttt{bvar}\ 0\ 1)) \\
& = & \texttt{let}\ \texttt{bvar}\ 0\ 1, \texttt{bvar}\ 1\ 0\ \texttt{in}\ \texttt{app}\ (\texttt{abs}\ \{2 \leftarrow x\}(\texttt{fvar}\ x))\ (\texttt{bvar}\ 0\ 1) \\
& = & \texttt{let}\ \texttt{bvar}\ 0\ 1, \texttt{bvar}\ 1\ 0\ \texttt{in}\ \texttt{app}\ (\texttt{abs}\ \texttt{bvar}\ 2\ 0)\ (\texttt{bvar}\ 0\ 1)
\end{array}
$$

Notice that the closed term coincides with u, although this is not always the case.

2.3 Local closure, free variables and substitution

The locally nameless syntax in Figure 1.b allows to build terms without a corresponding expression in *Exp*, e.g., the term `abs (bvar 1 5)` where index 1 does not refer to a binder in the term. Well-formed terms, i.e., those that correspond to expressions in *Exp*, are called *locally closed*.

To determine if a term is locally closed one should check that every bound variable in the term has valid indices, i.e., they refer to binders in the term. An easier method is to open with fresh names every abstraction and `let`-declaration in the term to be checked, and prove that no bound variable is ever reached. This is implemented with the *local closure* predicate `lc` (Figure 4).

Observe that we use cofinite quantification (as introduced by Aydemir et al. in [ACP+08]) in the rules for the binders, i.e., abstraction and `let`. Cofinite quantification is an elegant alternative to exist-fresh conditions and provides stronger induction principles. Proofs are simplified, as it is not required to define exactly the set of fresh names (examples of this are given in [Cha11]). The rule LC-ABS establishes that an abstraction is locally closed if there exists a finite set of names L such that, for any name x not in L, the term $t^{[x]}$ is locally closed. Similarly, in the rule LC-LET we write $\overline{x}^{|\overline{t}|} \notin L$ to indicate that the list of distinct names $\overline{x}$ of length $|\overline{t}|$ are not in the finite set L. For any list $\overline{x}$ satisfying this condition, the opening of each term in the list of local declarations, $\overline{t}^{\overline{x}}$, and of the term affected by these declarations, $t^{\overline{x}}$, are locally closed. We have overloaded the predicate `lc` to work both on terms and lists of terms; later on we will overload other predicates and functions similarly. We write $[t : \overline{t}]$ for the list with head t and tail $\overline{t}$. In the following, $[\,]$ represents the empty list, $[t]$ is a unitary list, and $+\!\!+$ is the concatenation of lists.

$$\text{LCK-BVAR}\ \frac{i < k \wedge j < \texttt{List.nth}\ i\ \overline{n}}{\texttt{lc\_at}\ k\ \overline{n}\ (\texttt{bvar}\ i\ j)} \qquad \text{LCK-APP}\ \frac{\texttt{lc\_at}\ k\ \overline{n}\ t \qquad \texttt{lc\_at}\ k\ \overline{n}\ v}{\texttt{lc\_at}\ k\ \overline{n}\ (\texttt{app}\ t\ v)}$$

$$\text{LCK-FVAR}\ \frac{}{\texttt{lc\_at}\ k\ \overline{n}\ (\texttt{fvar}\ x)} \qquad \text{LCK-LET}\ \frac{\texttt{lc\_at}\ (k+1)\ [|\overline{t}| : \overline{n}]\ [t : \overline{t}]}{\texttt{lc\_at}\ k\ \overline{n}\ (\texttt{let}\ \overline{t}\ \texttt{in}\ t)}$$

$$\text{LCK-ABS}\ \frac{\texttt{lc\_at}\ (k+1)\ [1 : \overline{n}]\ t}{\texttt{lc\_at}\ k\ \overline{n}\ (\texttt{abs}\ t)} \qquad \text{LCK-LIST}\ \frac{\texttt{List.forall}\ (\texttt{lc\_at}\ k\ \overline{n}\ \cdot)\ \overline{t}}{\texttt{lc\_at}\ k\ \overline{n}\ \overline{t}}$$

Figure 5: Local closure at level k

$$\begin{array}{lll} \texttt{fv}(\texttt{bvar}\ i\ j) & = \emptyset & (\texttt{bvar}\ i\ j)[z/y] = \texttt{bvar}\ i\ j \\ \texttt{fv}(\texttt{fvar}\ x) & = \{x\} & (\texttt{fvar}\ x)[z/y] = \begin{cases} \texttt{fvar}\ z & \text{if } x = y \\ \texttt{fvar}\ x & \text{if } x \neq y \end{cases} \\ \texttt{fv}(\texttt{abs}\ t) & = \texttt{fv}(t) & (\texttt{abs}\ t)[z/y] = \texttt{abs}\ t[z/y] \\ \texttt{fv}(\texttt{app}\ t\ v) & = \texttt{fv}(t) \cup \texttt{fv}(v) & (\texttt{app}\ t\ v)[z/y] = \texttt{app}\ t[z/y]\ v[z/y] \\ \texttt{fv}(\texttt{let}\ \overline{t}\ \texttt{in}\ t) & = \texttt{fv}(\overline{t}) \cup \texttt{fv}(t) & (\texttt{let}\ \overline{t}\ \texttt{in}\ t)[z/y] = \texttt{let}\ \overline{t}[z/y]\ \texttt{in}\ t[z/y] \end{array}$$

where $\texttt{fv}(\overline{t}) = \texttt{List.foldright}\ (\cdot \cup \cdot)\ \emptyset\ (\texttt{List.map}\ \texttt{fv}\ \overline{t})$
$\overline{t}[z/y] = \texttt{List.map}\ ([z/y]\cdot)\ \overline{t}$

Figure 6: Free variables and substitution

We define a new predicate that checks if indices in bound variables are valid from a given level: *t is closed at level k*, written $\texttt{lc\_at}\ k\ \overline{n}\ t$ (Figure 5), where k indicates the current binding depth. Since we are dealing with multibinders, we need to keep their size (1 in the case of an abstraction, n for a `let` with n local declarations). These sizes are collected in the list $\overline{n}$, so that $|\overline{n}|$ should be at least k. A bound variable $\texttt{bvar}\ i\ j$ is closed at level k if i is smaller than k and j is smaller than $\texttt{List.nth}\ i\ \overline{n}$. The list $\overline{n}$ is new with respect to [Cha11] because there the predicate `lc_at` is not defined for multiple binders.

The two approaches for local closure are equivalent, so that a term is locally closed if and only if it is closed at level 0: LC_IIF_LC_AT $\texttt{lc}\ t \Leftrightarrow \texttt{lc\_at}\ 0\ []\ t$.

Computing the *free variables* of a term t is very easy in the locally nameless representation, since bound and free variables are syntactically different. The set of free variables of $t \in LNExp$ is denoted as $\texttt{fv}(t)$, and it is defined in Figure 6.

A name x is said to be *fresh for a term t*, written $\texttt{fresh}\ x\ \texttt{in}\ t$, if x does not belong to the set of free variables of t. Similarly for a list of distict names $\overline{x}$:

$$\frac{x \notin \texttt{fv}(t)}{\texttt{fresh}\ x\ \texttt{in}\ t} \qquad \frac{\overline{x} \notin \texttt{fv}(t)}{\texttt{fresh}\ \overline{x}\ \texttt{in}\ t}$$

Substitution replaces a variable name by another name in a term. So that for $t \in LNExp$ and $z, y \in Id$, $t[z/y]$ is the term where z substitutes any occurrence of y in t (see Figure 6).

Under some conditions variable opening and closing are inverse operations. More precisely, opening a term with fresh names and closing it with the same names, produces the original term. Closing a locally closed term and then opening it with the same names gives back the term:

CLOSE_OPEN_VAR $\texttt{fresh}\ \overline{x}\ \texttt{in}\ t \Rightarrow {}^{\backslash \overline{x}}(t^{\overline{x}}) = t$ OPEN_CLOSE_VAR $\texttt{lc}\ t \Rightarrow ({}^{\backslash \overline{x}}t)^{\overline{x}} = t$

$$\textsc{Lam}\quad \Gamma : \lambda x.e \Downarrow \Gamma : \lambda x.e \qquad \textsc{App}\quad \frac{\Gamma : e \Downarrow \Theta : \lambda y.e' \qquad \Theta : e'[x/y] \Downarrow \Delta : w}{\Gamma : (e\ x) \Downarrow \Delta : w}$$

$$\textsc{Var}\quad \frac{\Gamma : e \Downarrow \Delta : w}{(\Gamma, x \mapsto e) : x \Downarrow (\Delta, x \mapsto w) : \hat{w}} \qquad \textsc{Let}\quad \frac{(\Gamma, \{x_i \mapsto e_i\}_{i=1}^n) : e \Downarrow \Delta : w}{\Gamma : \texttt{let}\ \{x_i = e_i\}_{i=1}^n\ \texttt{in}\ e \Downarrow \Delta : w}$$

Figure 7: Natural semantics

3 Natural semantics for lazy evaluation

The natural semantics defined by Launchbury [Lau93] follows a lazy strategy. Judgements are of the form $\Gamma : e \Downarrow \Delta : w$, that is, the expression $e \in \mathit{Exp}$ in the context of the heap Γ reduces to the value w in the context of the heap Δ. *Values* ($w \in \mathit{Val}$) are expressions in weak-head-normal-form (*whnf*). *Heaps* are partial functions from variables into expressions. Each pair (variable, expression) is called a *binding*, represented by $x \mapsto e$. During evaluation, new bindings may be added to the heap, and bindings may be updated to their corresponding computed values. The semantic rules are shown in Figure 7. The normalization of the λ-calculus, as mentioned in Section 2, simplifies the definition of the rules, although a renaming is still needed ($\hat{w}$ in VAR) to avoid name clashing. This renaming is justified by Barendregt's variable convention [Bar84].

Example 4 *Without the renaming in rule* VAR *heaps may end up binding a name more than once. Take for instance the evaluation of the expression*

$$e \equiv \texttt{let}\ x_1 = \lambda y.(\texttt{let}\ z = \lambda v.y\ \texttt{in}\ y), x_2 = (x_1\ x_3), x_3 = (x_1\ x_4), x_4 = \lambda s.s\ \texttt{in}\ x_2$$

in the context of the empty heap. The evaluation of e implies the evaluation of x_2, and then the evaluation of $(x_1 x_3)$. This application leads to the addition of $z \mapsto \lambda v.x_3$ to the heap. Eventually, the evaluation of x_3 implies the evaluation of $(x_1 x_4)$. Without a renaming of values, variable z is added again to the heap, now bound to $\lambda v.x_4$.

Theorem 1 in [Lau93] states that "every heap/term pair occurring in the proof of a reduction is *distinctly named*", but we have found that the renaming fails to ensure this property. At least, it depends on how much fresh is this renaming.

Example 5 *Let us evaluate in the context of the empty heap the following expression:*

$$e \equiv \texttt{let}\ x_1 = (x_2\ x_3), x_2 = \lambda z.\texttt{let}\ y = \lambda t.t\ \texttt{in}\ y, x_3 = \lambda s.s\ \texttt{in}\ x_1$$

$$\begin{array}{l}
\{\,\} : e \\
\textsc{Let}\ \Big|\ \{x_1 \mapsto (x_2\ x_3), x_2 \mapsto \lambda z.\texttt{let}\ y = \lambda t.t\ \texttt{in}\ y, x_3 \mapsto \lambda s.s\} : x_1 \\
\quad \textsc{Var}\ \Big|\ \{x_2 \mapsto \lambda z.\texttt{let}\ y = \lambda t.t\ \texttt{in}\ y, x_3 \mapsto \lambda s.s\} : (x_2\ x_3) \\
\quad\quad \textsc{App}\ \Big|\ \{x_2 \mapsto \lambda z.\texttt{let}\ y = \lambda t.t\ \texttt{in}\ y, x_3 \mapsto \lambda s.s\} : x_2 \\
\quad\quad\quad \textsc{Var}\ \Big|\ \{x_3 \mapsto \lambda s.s\} : \lambda z.\texttt{let}\ y = \lambda t.t\ \texttt{in}\ y \\
\quad\quad\quad\quad \textsc{Lam}\ \Big|\ \\
\quad\quad\quad\quad \vdots \qquad \{x_3 \mapsto \lambda s.s\} : \boxed{\lambda z.\texttt{let}\ y = \lambda t.t\ \texttt{in}\ y}
\end{array}$$

At this point the VAR*-rule requires to rename the value that has been obtained (which is highlighted in the square). Notice that x_1 is fresh in the actual heap/term pair, and thus could be*

chosen to rename y. This would lead later in the derivation to introduce x_1 twice in the heap. The solution is to consider the condition of freshness in the whole derivation. This notion has not been formally defined by Launchbury.

3.1 Locally nameless heaps

Before translating the semantic rules in Figure 7 to the locally nameless representation defined in Section 2, we have to show how *bindings* and *heaps* are represented in this notation. Bindings associate expressions to free variables, hence bindings are now pairs $(\texttt{fvar}\ x,t)$ with $x \in Id$ and $t \in LNExp$. To simplify, we just write $x \mapsto t$. In the following, we represent a heap $\{x_i \mapsto t_i\}_{i=1}^{n}$ as $(\overline{x} \mapsto \overline{t})$, with $|\overline{x}| = |\overline{t}| = n$. The set of locally-nameless-heaps is denoted as *LNHeap*.

The *domain* of a heap Γ, written $\texttt{dom}(\Gamma)$, collects the set of names that are bound in the heap:

$$\texttt{dom}(\emptyset) = \emptyset \qquad\qquad \texttt{dom}(\Gamma, x \mapsto t) = \texttt{dom}(\Gamma) \cup \{x\}$$

In a well-formed heap names are defined at most once and terms are locally closed. The predicate `ok` expresses that a heap is well-formed:

$$\text{OK-EMPTY}\quad \frac{}{\texttt{ok}\ \emptyset} \qquad\qquad \text{OK-CONS}\quad \frac{\texttt{ok}\ \Gamma \qquad x \notin \texttt{dom}(\Gamma) \qquad \texttt{lc}\ t}{\texttt{ok}\ (\Gamma, x \mapsto t)}$$

The function `names` returns the set of names that appear in a heap, i.e., the names occurring in the domain or in the right-hand side terms:

$$\texttt{names}(\emptyset) = \emptyset \qquad\qquad \texttt{names}(\Gamma, x \mapsto t) = \texttt{names}(\Gamma) \cup \{x\} \cup \texttt{fv}(t)$$

This definition can be extended to (heap: term) pairs: $\texttt{names}(\Gamma : t) = \texttt{names}(\Gamma) \cup \texttt{fv}(t)$
We also extend the freshness predicate:

$$\frac{\overline{x} \notin \texttt{names}(\Gamma : t)}{\texttt{fresh}\ \overline{x}\ \texttt{in}\ (\Gamma : t)}$$

Substitution of variable names is extended to heaps as follows:

$$\emptyset[z/y] = \emptyset \qquad\qquad (\Gamma, x \mapsto t)[z/y] = (\Gamma[z/y], x[z/y] \mapsto t[z/y])$$
$$\textit{where } x[z/y] = \begin{cases} z & \text{if } x = y \\ x & \text{otherwise} \end{cases}$$

3.2 Locally nameless semantics

Once the locally nameless syntax and the corresponding operations, functions and predicates have been defined, three steps are sufficient to translate an inductive definition on λ-terms from the named representation into the locally nameless notation (as it is explained in [Cha11]):

1. Replace the named binders, i.e., abstractions and let-declarations, with nameless binders by opening the bodies.
2. Cofinitely quantify the names introduced for variable opening.
3. Add premises to inductive rules in order to ensure that inductive judgements are restricted to locally closed terms.

$$\text{LNLAM}\quad \frac{\{\mathtt{ok}\ \Gamma\}\qquad \{\mathtt{lc}\ (\mathtt{abs}\ t)\}}{\Gamma : \mathtt{abs}\ t \Downarrow \Gamma : \mathtt{abs}\ t}$$

$$\text{LNVAR}\quad \frac{\Gamma : t \Downarrow \Delta : w \qquad \{x \notin \mathtt{dom}(\Gamma) \cup \mathtt{dom}(\Delta)\}}{(\Gamma, x \mapsto t) : (\mathtt{fvar}\ x) \Downarrow (\Delta, x \mapsto w) : w}$$

$$\text{LNAPP}\quad \frac{\Gamma : t \Downarrow \Theta : \mathtt{abs}\ u \qquad \Theta : u^{[x]} \Downarrow \Delta : w \qquad \{x \notin \mathtt{dom}(\Gamma) \Rightarrow x \notin \mathtt{dom}(\Delta)\}}{\Gamma : \mathtt{app}\ t\ (\mathtt{fvar}\ x) \Downarrow \Delta : w}$$

$$\text{LNLET}\quad \frac{\forall \overline{x}^{|\overline{t}|} \notin L \subseteq Id\ \ (\Gamma, \overline{x} \mapsto \overline{t}^{\overline{x}}) : t^{\overline{x}} \Downarrow (\overline{x} +\!\!+ \overline{z} \mapsto \overline{u}^{\overline{x}}) : w^{\overline{x}} \qquad \{\overline{y}^{|\overline{t}|} \notin L \subseteq Id\}}{\Gamma : \mathtt{let}\ \overline{t}\ \mathtt{in}\ t \Downarrow (\overline{y} +\!\!+ \overline{z} \mapsto \overline{u}^{\overline{y}}) : w^{\overline{y}}}$$

Figure 8: Locally nameless natural semantics

We apply these steps to the inductive rules for the lazy natural semantics given in Figure 7. These rules produce judgements involving λ-terms as well as heaps. Hence, we also add premises that ensure that inductive judgements are restricted to well-formed heaps. The rules using the locally nameless representation are shown in Figure 8. For clarity, in the rules we put in braces the side-conditions to distinguish them better from the judgements.

The main difference with the rules in Figure 7 is the rule LNLET. To evaluate $\mathtt{let}\ \overline{t}\ \mathtt{in}\ t$ the local terms in $\overline{t}$ are introduced in the heap, so that the body t is evaluated in this new context. Fresh names $\overline{x}$ are needed to open the local terms and the body. The evaluation of $t^{\overline{x}}$ produces a final heap and a value. Both are dependent on the names chosen for the local variables. The domain of the final heap consists of the local names $\overline{x}$ and the rest of names, say $\overline{z}$. The rule LNLET is cofinite quantified. As it is explained in [Cha11], the advantage of the cofinite rules over existential and universal ones is that the freshness side-conditions are not explicit. In our case, the freshness condition for $\overline{x}$ is *hidden* in the finite set L, which includes the names that should be avoided during the reduction. The novelty of our cofinite rule, compared with the ones appearing in [ACP+08] and [Cha11] (that are similar to the cofinite rules for the predicate `lc` in Figure 4), is that the names introduced in the (infinite) premises do appear in the conclusion too. Hence, in the conclusion of the rule LNLET we can replace the names $\overline{x}$ by any list $\overline{y}$ not in L.

The problem with explicit freshness conditions is that they are associated just to rule instances, while they should apply to the whole reduction proof. Take for instance the rule LNVAR. In the premise the binding $x \mapsto t$ does no longer belong to the heap. Therefore, a valid reduction for this premise may chose x as fresh (this corresponds to the problem shown in Example 5). We avoid this situation by requiring that x remains undefined in the final heap too. By contrast to the rule VAR in Figure 7, no renaming of the final value, that is w, is needed.

The new side-condition of rule LNAPP deserves an explanation. Suppose that x is undefined in the initial heap Γ, then we must avoid that x is chosen as a fresh name during the evaluation of t. For this reason we require that x is defined in the final heap Δ only if x was already defined in Γ. Notice how the body of the abstraction, that is u, is open with the name x. This is equivalent to the substitution of x for y in the body of the abstraction $\lambda y.e'$ (see rule APP in Figure 7).

A *regularity* lemma ensures that the judgements produced by this reduction system involve only well-formed heaps and locally closed terms.

REGULARITY $\quad \Gamma : t \Downarrow \Delta : w \Rightarrow \mathtt{ok}\ \Gamma \wedge \mathtt{lc}\ t \wedge \mathtt{ok}\ \Delta \wedge \mathtt{lc}\ w.$

Similarly, Theorem 1 in [Lau93] ensures that the property of being *distinctly named* is preserved by the rules in Figure 7. However, as shown in Example 5, the correctness of this result requires that freshness is relative to whole reduction proofs instead to the scope of rules.

A *renaming* lemma ensures that the evaluation of a term is independent of the fresh names chosen in the reduction process. Moreover, any name in the context can be replaced by a fresh one without changing the meaning of the terms evaluated in that context. In fact, reduction proofs for (heap: term) pairs are unique up to renaming of the variables defined in the context heap.

RENAMING $\quad \Gamma : t \Downarrow \Delta : w \wedge \texttt{fresh}\ y\ \texttt{in}\ (\Gamma : t) \wedge \texttt{fresh}\ y\ \texttt{in}\ (\Delta : w) \Rightarrow \Gamma[y/x] : t[y/x] \Downarrow \Delta[y/x] : w[y/x]$

In addition, the renaming lemma permits to prove an *introduction* lemma for the cofinite rule LNLET which establishes that the corresponding existential rule is admissible too.

LET_INTRO $\quad (\Gamma, \overline{x} \mapsto \overline{t^x}) : t^x \Downarrow (\overline{x} +\!\!+ \overline{z} \mapsto \overline{u^x}) : w^x \wedge \texttt{fresh}\ \overline{x}\ \texttt{in}\ (\Gamma : \texttt{let}\ \overline{t}\ \texttt{in}\ t)$
$\Rightarrow \Gamma : \texttt{let}\ \overline{t}\ \texttt{in}\ t \Downarrow (\overline{x} +\!\!+ \overline{z} \mapsto \overline{u^x}) : w^x$

This result, together with the renaming lemma, justifies that our rule LNLET is equivalent to Launchbury's rule LET used with normalized terms.

4 Implementation in Coq

We are currently extending the work of Charguéraud [Cha11] to our calculus, for later on formalizing Launchbury's semantics. In http://www.charguéraud.org/softs/ln/ one finds the Coq library TLC which provides the representation for variables and finite sets of variables, as well as the tactics that are needed to work with the locally nameless representation. The local definitions introduced by the `let`-declarations are the main difference between the syntax defined in [Cha11] and ours. In order to avoid some problems related with the induction principle, we have dropped for the moment the restriction in the application (see Figure 1.b).

Our first attempt was to define terms as follows:

```
Inductive trm : Set := | t_bvar : nat → nat → trm
                       | t_fvar : var → trm
                       | t_abs : trm → trm
                       | t_app : trm → trm → trm
                       | t_let : (list trm) → trm → trm.
```

Unfortunately, the induction principle defined by Coq does not go inside the list of terms that occurs in a local declaration. As explained by Bertot and Castéran in [BC04] we have to redefine the induction principle for mutual recursion. Therefore we redefine the terms as:

```
Inductive trm : Set := | t_bvar : nat → nat → trm
                       | t_fvar : var → trm
                       | t_abs : trm → trm
                       | t_app : trm → trm → trm
                       | t_let : L_trm → trm → trm
          with L_trm := | nil_Lt : L_trm
                       | cons_Lt : trm → L_trm → L_trm.
```

and afterwards we give the following induction scheme:

```
Scheme   trm_ind2   := Induction for trm Sort Prop
  with   L_trm_ind2 := Induction for L_trm Sort Prop.
```

When checking this induction principle the response of Coq is:

```
trm_ind2 : forall (P : trm -> Prop) (P0 : L_trm -> Prop),
           (forall n n0 : nat, P (t_bvar n n0)) ->
           (forall v : var, P (t_fvar v)) ->
           (forall t : trm, P t -> P (t_abs t)) ->
           (forall t : trm, P t -> forall t0 : trm, P t0 -> P (t_app t t0)) ->
           (forall l : L_trm, P0 l -> forall t : trm, P t -> P (t_let l t)) ->
           P0 nil_Lt ->
           (forall t : trm, P t -> forall l : L_trm, P0 l -> P0 (cons_Lt t l)) ->
           forall t : trm, P t
```

There are two properties: P and $P0$. The former refers to terms and the latter to list of terms.

We have extended the recursive functions to open and close at level k. As a sample, we show here the opening at level k. Observe how the definition of this function reproduces the two-layers structure (terms and list of terms):

```
Fixpoint open_rec (k : nat) (vs : list var) (t : trm) {struct t} : trm :=
  match t with
   | t_bvar i j    ⇒ if (andb (beq_nat k i) (blt_nat j (length vs)))
                     then (t_fvar(nth j vs a)) else (t_bvar i j)
   | t_fvar x      ⇒ t_fvar x
   | t_abs t1      ⇒ t_abs (open_rec (S k) vs t1)
   | t_app t1 t2   ⇒ t_app (open_rec k vs t1) (open_rec k vs t2)
   | t_let ts t    ⇒ t_let (open_L_rec (S k) vs ts) (open_rec (S k) vs t)
  end
with open_L_rec (k : nat) (vs : list var) (ts : L_trm) {struct ts} : L_trm :=
  match ts with
   | nil_Lt        ⇒ nil_Lt
   | cons_Lt t ts  ⇒ cons_Lt (open_rec k vs t) (open_L_rec k vs ts)
  end.
```

The boolean function `beq_nat` checks if two given numbers are equal. Analogously, `blt_nat` indicates if a natural number is less or equal than a another. As expected, `length` returns the length of a list and `nth j vs a` returns the j-nth element of the list `vs`, where `a` is the default value for an index out of range (although in our definition this default value is never returned).

Now the definition of the opening operation is:

```
Definition open t vs := open_rec 0 vs t.
```

The locally closed predicate, which involves cofinite quantification over a list of identifiers, is inductively defined as:

```
Inductive  lc : trm -> Prop :=
         | lc_var : forall x, lc (t_fvar x)
         | lc_app : forall t1 t2, lc t1 -> lc t2 -> lc (t_app t1 t2)
         | lc_abs : forall L t, (forall x, x ∉ L -> lc (open t (cons x nil))) -> lc (t_abst)
         | lc_let : forall L t ts, (forall xs, xs ∉ L -> (lc_list (opens (cons_Lt t ts) xs)))
                    -> lc (t_let ts t)
    with lc_list : L_trm -> Prop :=
         | lc_list_nil : lc_list (nil_Lt)
         | lc_list_cons : forall t ts, lc t -> lc_list ts -> lc_list(cons_Lt t ts).
```

where `opens` is the generalization of the function `open` to lists of terms.

We use the induction principle `trm_ind2` defined above to prove the

```
Lemma lc_subs_lc : forall t, lc t -> lc (subst y z t).
```

To apply this principle we need to specify the property `P0` over lists of terms. In this case the tactic to be used is: `elim t using trm_ind2 with (P0 := fun ts : L_trm => lc_list ts -> lc_list (substL y z ts))`.

At present, we are working on the definition of heaps and the corresponding predicates.

5 Related work

In order to avoid α-conversion, we first considered a nameless representation like the de Bruijn notation [dB72], where variable names are replaced by natural numbers. But this notation has several drawbacks. First of all, the de Bruijn representation is hard to read for humans. Even if we intend to check our results with a proof assistant, human readability helps intuition. Moreover, the de Bruijn notation does not have a good way to handle free variables, which are represented by indices, alike to bound variables. This is a serious weakness for our application. Launchbury's semantics uses context heaps that collect the bindings for the free variables that may occur in the term under evaluation. Any change in the domain of a heap, i.e., adding or deleting a binding, would lead to a shifting of the indices, thus complicating the statement and proof of results. Therefore, we prefer the more manageable locally nameless representation, where bound variable names are replaced by indices but free variables keep their names. This mixed notation combines the advantages of both named and nameless representations. On the one hand, α-conversion is avoided all the same. On the other hand, terms stay readable and easy to manipulate.

There exists in the literature different proposals for a locally nameless representation, and many works using these representations. Charguéraud offers in [Cha11] a brief survey on these works, that we recommend to the interested reader.

Launchbury implicitly assumes Barendregt's variable convention [Bar84] twice in [Lau93]. First when he defines his operational semantics only for normalized λ-terms (i.e., every binder in a term binds a distinct name, which is also distinct from any free variable); and second, when he requires a (fresh) renaming of the values in the rule VAR (Figure 7). Urban, Berghofer and Norrish propose in [UBN07] a method to strengthen an induction principle (corresponding to some inductive relation), so that the variable convention comes already built in the principle. Unfortunately, we cannot apply these ideas to Launchbury's semantics, because the semantic rules do not satisfy the conditions that guarantee the *variable convention compatibility*, as described in [UBN07]. In fact, as we have already pointed out, Launchbury's Theorem 1 (in [Lau93]) is correct only if the renaming required in each application of the rule VAR is fresh in the whole reduction proof. Therefore, we cannot use directly Urban's nominal package for Isabelle/HOL [Urb08] (including the recent extensions for general bindings described in [UK10]).

Nevertheless, Urban et al. achieve the "inclusion" of the variable convention in an induction principle by adding to each induction rule a side condition which expresses that the set of *bound* variables (those that appear in a binding position in the rule) are fresh in some *induction context* ([UBN07]). Furthermore, this context is required to be finitely supported. This is closely related to the cofinite quantification that we have used for the rule LNLET in Figure 8. Besides, a

condition to ensure the variable convention compatibility is the *equivariance* of the functions and predicates occurring in the induction rules. Equivariance is a notion from nominal logic [Pit03]. A relation is equivariant if it is preserved by permutation of names. Although we have not proven that the reduction relation defined by the rules in Figure 8 is equivariant, our *renaming lemma* establishes a similar result, that is, the reduction relation is preserved by (fresh) renaming.

Nowadays there is a wide range of proof checkers and theorem provers. Wiedijk has compiled in http://www.cs.ru.nl/ freek/digimath/ a large list of these computer systems, classified by a number of categories. Focusing in proof assistants, Barendregt and Geuvers give in [BG01] an interesting discussion and comparison of the most widely-known proof assistants, that it is still effective. We also recommend the lecture of the paper by Geuvers [Geu09] on the main concepts and history of proof assistants. We have already commented in the introduction our reasons for using Coq [Ber06], that is, dependent and inductive types, tactics, and proof by reflection.

6 Present and future

We have used a more modern approach to binding, i.e., a locally nameless representation for the λ-calculus extended with mutually recursive local declarations. With this representation the reduction rule for local declarations implies the introduction of fresh names. We have used neither an existential nor a universal rule for this case. Instead, we have opted for a cofinite rule as introduced by Aydemir et al. in [ACP+08]. Freshness conditions are usually considered in each rule individually. Nevertheless, this technique produces name clashing when considering whole reduction proofs. A solution might be to decorate judgements with the set of forbidden names and indicate how to modify this set during the reduction process (this approach has been taken by Sestoft in [Ses97]). However, this could be too restrictive in many occasions. Besides, existential rules are not easy to deal with because each reduction is obtained just for one specific list of names. If any of the names in this list causes a name clashing with other reduction proofs, then it is cumbersome to demonstrate that an alternative reduction for a fresh list does exist. Cofinite quantification solves this problem because in a single step reductions are guaranteed for an infinite number of lists of names. Nonetheless, our (LET_INTRO) lemma guarantees that a more conventional exists-fresh rule is correct in our reduction system too.

The cofinite quantification that we have used in our semantic rules is more complex than those in [ACP+08] and [Cha11]. The cofinite rule LNLET in Figure 8 introduces quantified variables in the conclusion as well, as the latter depends on the chosen names.

Compared to Launchbury's semantic rules, our locally nameless rules include several extra side-conditions. Some of these require that heaps and terms are well-formed (e.g., in LNLAM). Others express restrictions on the choice of fresh names, so that together with the cofinite quantification, fix the problem with the renaming in rule VAR that we have shown in Example 5.

Our locally nameless semantics satisfies a *regularity lemma* which ensures that every term and heap involved in a reduction proof is well-formed, and a *renaming lemma* which indicates that the choice of names (free variables) is irrelevant as long as they are fresh enough. A heap may be seen as a multiple binder. Actually, the names defined (bound) in a heap can be replaced by other names, provided that terms keep their meaning in the context represented by the heap. Our renaming lemma ensures that whenever a heap is renamed with fresh names, reduction proofs

are preserved. This renaming lemma is essential in rule induction proofs for some properties of the reduction system. More concretely, when one combines several reduction proofs coming from two or more premises in a reduction rule (for instance, in rule LNAPP in Figure 8).

So far, our contributions comprises the following:

1. A locally nameless representation of a λ-calculus with recursive local declarations;
2. A locally nameless version of Launchbury's natural semantics for lazy evaluation;
3. A new version of cofinite rules where the variables quantified in the premises do appear in the conclusion too;
4. A way to guarantee Barendregt's variable convention by redefining Launchbury's semantic rules with cofinite quantification and extra side-conditions;
5. A set of interesting properties of our reduction system, including the regularity, the introduction and the renaming lemmas; and
6. An *ongoing* implementation in Coq of Launchbury's semantics.

In the future we will use the implementation in Coq to prove formally the equivalence of Launchbury's natural semantics with the alternative version given also in [Lau93]. As we mentioned in Section 1, this alternative version differs from the original one in the introduction of indirections during β-reduction and the elimination of updates. We have already started with the definition (using the locally nameless representation) of two intermediate semantics, one introducing indirections and the other without updates. We are investigating equivalence relations between heaps obtained by each semantics, which makes able to prove the equivalence of the original natural semantics and the alternative semantics through these intermediate semantics.

References

[ACP+08] B. E. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, S. Weirich. Engineering formal metatheory. In *ACM Symposium on Principles of Programming Languages (POPL'08)*. Pp. 3–15. ACM Press, 2008.

[AGD] Agda. `http://wiki.portal.chalmers.se/agda/`.

[Bar84] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics 103. North-Holland, 1984.

[BC04] Y. Bertot, P. Castéran. *Interactive Theorem Proving and Program Development. Coq' Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[Ber06] Y. Bertot. Coq in a Hurry. *CoRR* abs/cs/0603118, 2006.

[BG01] H. Barendregt, H. Geuvers. Proof-assistants using dependent type systems. In Robinson and Voronkov (eds.), *Handbook of automated reasoning*. Pp. 1149–1238. Elsevier Science Publishers B. V., 2001.

[BKT00] C. Baker-Finch, D. King, P. W. Trinder. An Operational Semantics for Parallel Lazy Evaluation. In *ACM-SIGPLAN International Conference on Functional Programming (ICFP'00)*. Pp. 162–173. ACM Press, 2000.

[dB72] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae* 75(5):381–392, 1972.

[Cha11] A. Charguéraud. The Locally Nameless Representation. *Journal of Automated Reasoning*, pp. 1–46, 2011.

[EM07] M. van Eekelen, M. de Mol. *Reflections on Type Theory, λ-calculus, and the Mind. Essays dedicated to Henk Barendregt on the Occasion of his 60th Birthday*. Chapter Proving Lazy Folklore with Mixed Lazy/Strict Semantics, pp. 87–101. Radboud University Nijmegen, 2007.

[Geu09] H. Geuvers. Proof assistants: History, ideas and future. *Sadhana* 34(1):3–25, 2009.

[Lau93] J. Launchbury. A Natural Semantics for Lazy Evaluation. In *ACM Symposium on Principles of Programming Languages (POPL'93)*. Pp. 144–154. ACM Press, 1993.

[NPW02] T. Nipkow, L. C. Paulson, M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer, 2002.

[Pit03] A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation* 186(2):165–193, 2003.

[Ses97] P. Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming* 7(3):231–264, 1997.

[SHO10] L. Sánchez-Gil, M. Hidalgo-Herrero, Y. Ortega-Mallén. *Trends in Functional Programming*. Volume 10, chapter An Operational Semantics for Distributed Lazy Evaluation, pp. 65–80. Intellect, 2010.

[SHO12] L. Sánchez-Gil, M. Hidalgo-Herrero, Y. Ortega-Mallén. A locally nameless representation for a natural semantics for lazy evaluation. Technical report 01/12, Dpt. Sistemas Informáticos y Computación. Universidad Complutense de Madrid, 2012. http://maude.sip.ucm.es/eden-semantics/.

[UBN07] C. Urban, S. Berghofer, M. Norrish. Barendregt's Variable Convention in Rule Inductions. In *Proceedings of the 21st International Conference on Automated Deduction: Automated Deduction*. Pp. 35–50. Springer-Verlag, 2007.

[UK10] C. Urban, C. Kaliszyk. General Bindings and Alpha-Equivalence in Nominal Isabelle. In *Proceedings of the 20th European Symposium on Programming*. Pp. 480–500. Springer-Verlag, 2011.

[Urb08] C. Urban. Nominal Techniques in Isabelle/HOL. *Journal of Automatic Reasoning* 40(4):327–356, 2008.

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): Actas de las "XII Jornadas sobre Programación y Lenguajes (PROLE'2012), Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

Improving the Performance of FD Constraint Solving in a CFLP System *

Ignacio Castiñeiras[1] **and Fernando Sáenz-Pérez**[2]

[1] ncasti@fdi.ucm.es
Dept. Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain

[2] fernan@sip.ucm.es, http://www.fdi.ucm.es/profesor/fernan
Dept. Ingeniería del Software e Inteligencia Artificial
Universidad Complutense de Madrid, Spain

Abstract: Constraint Functional Logic Programming (CFLP) integrates lazy narrowing with constraint solving. It provides a high modeling abstraction, but its solving performance can be penalized by lazy narrowing and solver interface surcharges. As for real-world problems most of the solving time is carried out by solver computations, the system performance can be improved by interfacing state-of-the-art external solvers with proven performance. In this work we consider the CFLP system $\mathcal{TOY}(\mathcal{FD})$, implemented in SICStus Prolog and supporting Finite Domain ($\mathcal{FD}$) constraints by using its underlying Prolog $\mathcal{FD}$ solver. We present a scheme describing how to interface an external CP($\mathcal{FD}$) solver to $\mathcal{TOY}(\mathcal{FD})$, and easily adaptable to other Prolog CLP or CFLP systems. We prove the scheme to be generic enough by interfacing Gecode and ILOG solvers, and we analyze the new performance achieved.

Keywords: Constraint Functional Logic Programming, Finite Domains Constraint Solving, Solver Integration

1 Summary

Planning and scheduling problems are present in manufacturing and service industries. Their modeling is complex and susceptible to ongoing changes, and their solving implies much computational effort. Mathematical Programming and Heuristics methods can be used to tackle them, but their constraint-oriented nature makes Constraint Programming a suitable approach, as those problems can be modeled and solved as Constraint Satisfaction Problems (CSP) over Finite Domains ($\mathcal{FD}$). Focusing on this area, the Constraint Logic Programming (CLP($\mathcal{FD}$)) and Constraint Functional Logic Programming (CFLP($\mathcal{FD}$)) languages provide more expressivity than the Constraint Programming (CP($\mathcal{FD}$)) ones, due to the features provided by the logic and functional components. Hovewer, CLP($\mathcal{FD}$) and CFLP($\mathcal{FD}$) decrease CP($\mathcal{FD}$) efficiency, as they intermix constraint solving with SLD resolution and lazy narrowing (resp.) Overhead

* This work has been partially supported by the Spanish projects TIN2008-06622-C03-01, UCM-BSCH-GR58/08-910502, and S2009TIC-1465

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): PROLE'2012, pp. 31-32, ISBN:978-84-15487-27-2.
Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

comes from interfacing the constraint solver to the system and coordinating it. In this context, solving efficiency of CLP($\mathcal{FD}$) and CFLP($\mathcal{FD}$) systems should be improved to become a valid alternative to CP($\mathcal{FD}$) for tackling real-world problems. As for those problems most of the solving time is carried out by solver computations, the best way to improve solving efficiency is by replacing underlying solvers by other state-of-the-art external solvers with proven performance.

The main contribution of this paper is to present a generic scheme for interfacing CP($\mathcal{FD}$) libraries with C++ API to CLP($\mathcal{FD}$) and CFLP($\mathcal{FD}$) systems implemented in Prolog, with the aim of improving their constraint solving performance. It focuses on interfacing Gecode 3.7.0 [1] and IBM ILOG Solver 6.8 [2] to the CFLP($\mathcal{FD}$) system $\mathcal{TOY}(\mathcal{FD})$ [FHSV07], implemented in SICStus Prolog [3]. As a starting point to the implementation and a reference of the system performance we use $\mathcal{TOY}(\mathcal{FD}s)$ [FHSV07], which uses the SICStus underlying $\mathcal{FD}$ constraint solver `clpfd` to provide a clean and elegant interface between system and solver.

Interfacing external solvers increases the interface complexity, by including the management of: (1) Communication between the system and solver different language components, (2) Different representation of variables, constraints and types, (3) Explicit backtracking of the solver, (4) Support of multiple search procedures interleaved with constraint posting, and (5) Incremental and batch propagation modes (those last three points are needed to fit the CP($\mathcal{FD}$) solver to the modeling reasoning framework supported by the system.) The scheme being presented for interfacing external solvers includes all these features, and is proved to be generic enough, as both the Gecode and ILOG solvers are interfaced (giving rise to the new system versions $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$, resp.) by simply matching the steps described in the scheme.

The paper applies the three $\mathcal{TOY}(\mathcal{FD})$ versions to different instances (with different orders-of-magnitude solving times) of the real-life employee timetabling problem described in [CS11], obtaining the following conclusions: (I) The approach of enhancing the $\mathcal{TOY}(\mathcal{FD})$ performance by focusing on its $\mathcal{FD}$ solver is encouraging. As instances scale, the $\mathcal{FD}$ solver computation time represents the (80%-98%) of the total elapsed time. (II) The performance of $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ clearly outperforms $\mathcal{TOY}(\mathcal{FD}s)$ between 2-4 times. However, for small problems that do not require big $\mathcal{FD}$ solver computations, $\mathcal{TOY}(\mathcal{FD}s)$ is preferable, as the surcharges for interfacing a C++ external solver penalize the $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ performance. (III) The use of batch propagation mode is also encouraged, as in some cases it improves the $\mathcal{TOY}(\mathcal{FD})$ performance.

Bibliography

[CS11] I. Castiñeiras, F. Sáenz-Pérez. A CFLP Approach for Modeling and Solving a Real Life Employee Timetabling Problem. In *COPLAS'11, 63–70*. 2011.

[FHSV07] A. J. Fernández, T. Hortalá-González, F. Sáenz-Pérez, R. del Vado-Vírseda. Constraint Functional Logic Programming over Finite Domains. *TPLP* 7(5): 537–582, 2007.

[1] http://www.gecode.org/
[2] http://www-947.ibm.com/support/entry/portal/Overview/Software/WebSphere/IBM_ILOG_CP
[3] http://www.sics.se/isl/sicstus

Tecnología funcional en aplicaciones de televisión interactiva: acceso a redes sociales desde Synthetrick

David Duque[1], **Laura M. Castro**[2*]

[1] david.duque@udc.es
[2] lcastro@udc.es
Grupo MADS – Departamento de Computación
Universidade da Coruña (A Coruña, España)

Abstract: En este artículo presentamos una alternativa, basada en el uso de tecnología funcional, al desarrollo e implantación tradicionales de contenidos interactivos para televisión digital. Lo hacemos presentando el desarrollo de *Synthsocial*, un servicio interactivo de entretenimiento que ofrece acceso a diferentes redes sociales desde la plataforma *Synthetrick*.

En su concepción original, los servicios interactivos se ejecutan en los *set-top boxes* (STB), dispositivos instalados por el operador en el hogar del cliente. Con este modelo, el flujo de vídeo se genera y reproduce de forma local directamente en los STB del lado del cliente. Esto presenta graves problemas de dependencia de la plataforma específica y de interoperabilidad con otras distintas, entre otros.

Como solución a dichos problemas, la plataforma *Synthetrick* propone el uso de un servidor de recursos sintéticos. Con este nuevo enfoque, las aplicaciones se ejecutan en el lado del servidor, donde se genera el flujo de vídeo (conocido como vídeo sintético) que posteriormente es recibido y directamente reproducido por el STB. Los STB, a su vez, interactúan con el servidor enviándole las entradas introducidas por el usuario para que sean tenidas en cuenta en la generación del contenido que se le está presentando al consumidor.

Utilizando como caso de estudio el servicio *Synthsocial*, no sólo mostramos las ventajas de esta aproximación, sino que recomendamos y discutimos la arquitectura óptima para este tipo de innovadores servicios interactivos.

Keywords: televisión interactiva, servicios interactivos, vídeo sintético, programación funcional, erlang, synthetrick

1. Introducción

Las tecnologías de la información y de las comunicaciones (TIC) se extienden cada vez a más ámbitos de la sociedad, con una importante tasa de impacto en numerosos y crecientes aspectos de nuestra vida, cambiando hábitos de consumo, de entretenimiento, maneras de relacionarse, etc.

* Trabajo parcialmente financiado por el proyecto EU FP7 OptiBand (grant n. 248495), y MICIN TIN2010-20959.

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): PROLE'2012, pp. 33-47, ISBN:978-84-15487-27-2.
Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

Un claro ejemplo está en la llegada de la *televisión digital* a los hogares de los consumidores y la consecuente demanda de nuevos contenidos, impensables con la televisión tradicional. Entre ellos encontramos los llamados *servicios interactivos*, que permiten complementar la experiencia de usuario al hacerle posible acceder cómodamente desde su televisor, sólo con su mando a distancia, a múltiples contenidos de distinta naturaleza, de interés público, comercial o de entretenimiento.

Según la concepción tradicional del desarrollo e implantación de servicios interactivos, éstos se ejecutan de forma local en los propios *set-top boxes* (STB). Es decir, es el propio STB el encargado de ejecutar el servicio interactivo y de construir la versión final del vídeo que se va a presentar al usuario a partir de los datos obtenidos. Por este motivo, presentan un grave problema asociado a su gran dependencia de la plataforma hardware/software específica para la que se desarrollan y la consecuente incompatibilidad para interoperar con otras distintas.

Como solución, la plataforma de desarrollo *Synthetrick* [htt12b] permite el uso de un servidor de recursos sintéticos. Con este nuevo enfoque, las aplicaciones se ejecutan en el lado del servidor, donde se genera el flujo de vídeo que es recibido y reproducido directamente por el STB, que a su vez puede interactuar con el servidor enviándole las entradas introducidas por el usuario para que sean tenidas en cuenta en la generación del contenido que se le está presentando al consumidor. De esta manera, el vídeo se envía a los STBs ya completamente generado, por lo que éstos tan sólo deben descodificar y reproducir la señal recibida para presentarla al cliente. Esta innovadora idea aporta una serie de importantes ventajas al proceso de creación, implantación y mantenimiento de aplicaciones para televisión interactiva: reducción de costes, facilidad de desarrollo, interoperabilidad entre distintas plataformas, etc.

Por otro lado, un aspecto de la vida cotidiana que también está cambiando en los últimos tiempos es el que afecta a las relaciones sociales. La generalización del uso de Internet, ampliamente presente no sólo en los hogares y lugares de trabajo, sino también en los distintos dispositivos móviles de uso general (teléfonos, *tablets*...), ha provocado el auge de las *redes sociales*, medios de comunicación en línea que, en múltiples formatos y versiones, forman ya parte de la vida diaria de millones de personas en todo el mundo.

Aunando estos dos aspectos, servicios interactivos y redes sociales, en este artículo presentamos la creación de *Synthsocial*, un servicio interactivo de entretenimiento basado en la innovadora idea implementada por *Syntheractive* [htt12a] en su plataforma *Synthetrick*, que ilustramos mediante el ofrecimiento de un acceso amigable y unificado a diferentes redes sociales. Se pretende, por tanto, utilizando el acceso a redes sociales como caso de estudio, mostrar la construcción de una aplicación interactiva para televisión digital que aprovecha las novedosas características de esta nueva manera de hacer contenidos televisivos con soporte interactivo, cuya arquitectura subyacente se basa en tecnología funcional distribuida. Así, este caso de estudio da la oportunidad de evaluar las ventajas y madurez de la tecnología de *Synthetrick* de una manera realista y completa.

2. Plataforma *Synthetrick*

La plataforma *Synthetrick* es un *framework* para el desarrollo de servicios interactivos, denominados en este caso *servicios Syntheractive*. Proporciona un *kit* de desarrollo (SDK) basado

en C++ [Str09] como lenguaje de programación y que utiliza OpenSceneGraph (OSG) [WQ10] como motor de gráficos en 3D.

Synthetrick aporta una serie de importantes ventajas frente a un desarrollo tradicional ligado a la plataforma hardware y al soporte software proporcionado por un STB concreto, que pueden analizarse desde distintos puntos de vista: el de las operadoras de televisión, el de los desarrolladores de aplicaciones y el de los usuarios finales.

Desde la perspectiva de una operadora, utilizar vídeo sintético generado incrementa el tiempo de vida del hardware que se encuentra en uso en los hogares de sus clientes ya que, gracias a la ejecución de los servicios en el lado del servidor, antiguos STB pueden seguir siendo válidos para acceder a las últimas aplicaciones interactivas con gráficos 3D de alta calidad. Dicho de otra manera, el operador no se ve obligado a reemplazar los STB que ha proporcionado a sus clientes por otros mucho más costosos y con capacidades de última generación para permitir que sus usuarios tengan acceso a las últimas novedades en términos visuales y de interactividad.

Desde el punto de vista de los desarrolladores, el uso de este *framework* elimina los límites físicos y computacionales de los STB, haciendo posible el desarrollo de aplicaciones interactivas que hagan uso de características de un entorno de alta tecnología (proporcionadas por *Synthetrick*) sin tener que preocuparse en ningún caso de las posibles restricciones impuestas por el hardware del que pueda disponer el usuario. También se suprime el problema de interoperabilidad de los servicios interactivos. La aplicación se desarrolla una única vez pero puede ser implantada en plataformas hardware diversas: STBs, *Smart TVs*, teléfonos móviles o cualquier otro dispositivo con la simple capacidad de reproducción de vídeo. Del mismo modo, el servicio interactivo puede distribuirse por diferentes medios: IPTV, OTT, DVB-C, DVB-T, etc. Al ejecutarse en el servidor, los servicios interactivos basados en *Synthetrick* pueden proporcionar, además de gráficos 3D de alta calidad, características avanzadas impensables en servicios clásicos basados en otros modelos, como pueden ser la comunicación entre sesiones (que permite la creación de aplicaciones multi-usuario) o el almacenamiento persistente de gran capacidad. La ejecución en el servidor también hace posible ofrecer diferentes modos de visualización sin necesidad de desarrollo adicional, simplemente habilitando los diferentes modos disponibles en *Synthetrick* (HD, 3D estereoscópico...) para cada aplicación. Todas estas ventajas, naturalmente, repercuten significativamente en la reducción del tiempo de desarrollo.

Por último, el usuario final también se ve beneficiado por el uso de esta nueva tecnología que le permite acceder a novedosas y potentes aplicaciones interactivas en 3D en su televisor sin necesidad de cambiar su descodificador (STB). De hecho, ni siquiera es necesario uno si se dispone de algún otro dispositivo con capacidad de reproducción de vídeo, como una *Smart TV* o un teléfono móvil de última generación.

Synthetrick es además una plataforma altamente escalable cuyo núcleo está implementado utilizando tecnología funcional. Diseñada para ejecutarse sobre un *cluster* en el que cada nodo dispone de múltiples GPUs (*Graphics Processing Unit*), es capaz de soportar la interacción de miles de usuarios simultáneos [PGA09, TGMR11]. Cada uno de los nodos de *Synthetrick* cuenta con un planificador a corto plazo que permite asegurar la multiplexación y transmisión en tiempo real. El aprovechamiento de la relación prestaciones/coste de las GPUs de consumo ha permitido a esta plataforma conseguir una drástica reducción del coste por flujo, a la vez que presenta una alta optimización del sistema de codificación y renderizado basado en GPUs. Además de este planificador a corto plazo, otro a largo plazo equilibra la carga y permite aplicar diferentes

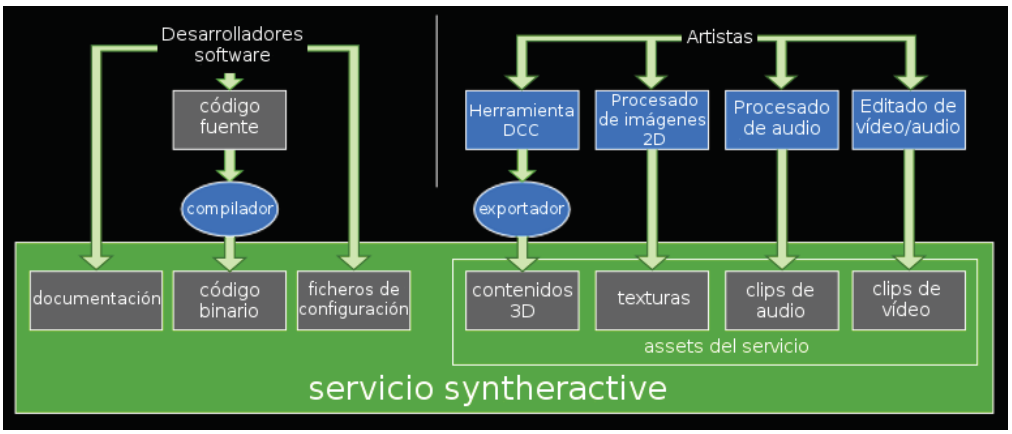


Figura 1: Creación de servicios interactivos con *Synthetrick*

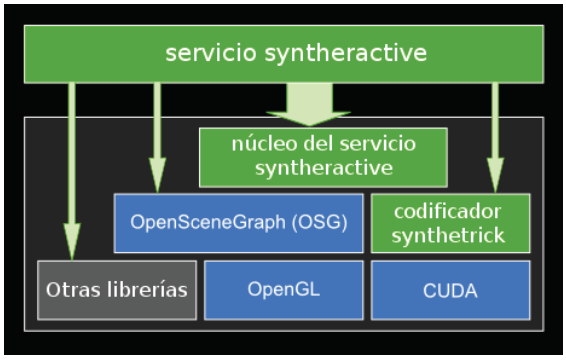


Figura 2: Ejecución de servicios interactivos sobre *Synthetrick*

políticas de distribución. Por último, posee un sistema de control distribuido que se encarga de la admisión, gestión de cuentas y recursos, planificación, supervisión y tolerancia a fallos.

Los servicios interactivos en *Synthetrick* son módulos independientes que se pueden cargar o descargar de cualquier nodo. Están compuestos por binarios, ficheros de configuración, *assets* (modelos 3D, texturas, animaciones...), documentación y código fuente. La Figura 1 muestra la arquitectura general de estos servicios. En esta figura se plasma el proceso de desarrollo de contenidos interactivos siguiendo el enfoque de esta tecnología. Puede verse cómo la labor de los desarrolladores software está completamente separada de la de los artistas. De esta manera, cada colectivo puede realizar su trabajo de la mejor forma posible (haciendo uso de las herramientas que prefieran, sin que tengan que ceñirse a ninguna restricción por parte de la plataforma...). En particular, los desarrolladores sólo tendrán que preocuparse de obtener un diseño y una implementación del servicio óptimos, además de aportar toda la documentación y ficheros de configuración necesarios. Finalmente, combinando el resultado de ambos cometidos, el conjunto es el servicio interactivo preparado para su instalación y ejecución en el servidor de recursos de la plataforma.

Por otro lado, la estructura seguida en la ejecución de los servicios interactivos en *Synthetrick* y la forma en la que el sistema se apoya en OSG y otras librerías [SK10] están representadas en la Figura 2. La idea es que los servicios *Syntheractive* están formados por un núcleo principal donde se define su comportamiento, incluyendo su inicialización y su reacción ante los distintos eventos que pueden darse en el sistema. El *kit* de desarrollo para esta plataforma está basado en OSG, que a su vez es una capa de abstracción sobre OpenGL. El codificador *Synthetrick* abstrae la interacción de la parte gráfica del servicio con CUDA, un conjunto de herramientas de desarrollo de NVIDIA que permiten codificar algoritmos en GPU. Finalmente, cada servicio puede necesitar de funcionalidades de otras librerías. Por ejemplo, como se detalla en la sección 3.1.3, *Synthsocial* hace uso de *erl interface* [Eri12] para posibilitar el intercambio de datos entre sus subsistemas.

3. Desarrollo de un servicio interactivo

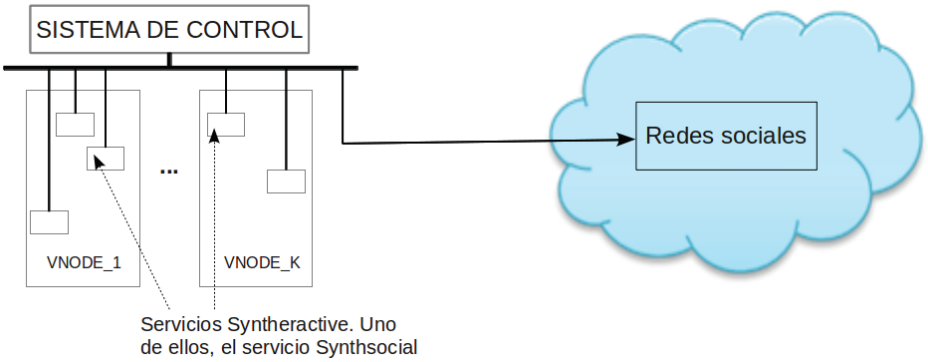
Como caso de estudio del uso real de *Synthetrick*, elegimos el desarrollo de un servicio interactivo para el acceso integrado a diferentes redes sociales. Mediante el desarrollo de este caso de estudio, pretendemos corroborar las características de la plataforma y comprobar su madurez, al mismo tiempo que ofrecemos una metodología reutilizable al desarrollo de servicios *Syntheractive* de la que poder abstraer elementos comunes, más allá de las funcionalidades proporcionadas por el propio SDK.

La Figura 3a presenta la arquitectura general de los servicios interactivos *Syntheractive*, gestionados de manera global por el sistema de control de *Synthetrick* y, gracias a ello, ejecutados de manera transparente en uno o varios de los diferentes nodos disponibles. Nuestro caso de estudio es uno de esos servicios, cuya estructura se muestra de forma más detallada en la Figura 3b. En ella, se refleja su organización en una serie de intermediarios (*proxies Synthsocial*) encargados de proporcionar y controlar el acceso a la información de las redes sociales.

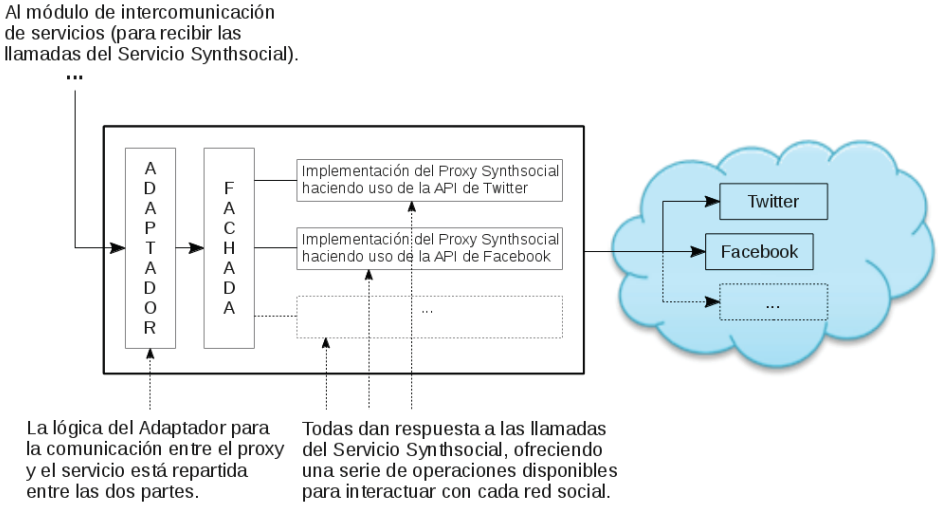
El sistema de gestión e intermediación de acceso a las redes sociales se ha desarrollado utilizando el lenguaje de programación Erlang [Arm07, CT09, LMC10] y presenta una estructura con una única fachada común que uniformiza el acceso a todas las redes sociales desde la interfaz del servicio interactivo. Internamente, el sistema modela un *proxy* concreto para cada red social (Facebook [Gra08, Gol08], Twitter [Mak09], etc.), de manera que las peticiones recibidas del usuario se encaminan al servidor adecuado para tratarlas, especializado éste en el uso de la API (*Application Programming Interface*) concreta ofrecida por la red social correspondiente. Esto permite dotar al sistema de la flexibilidad suficiente para posibilitar el futuro soporte de nuevas redes sociales con el mínimo esfuerzo.

3.1. *Synthsocial*: acceso a redes sociales

Synthsocial es el nombre con el que se ha bautizado al servicio interactivo que nos sirve de caso de estudio. Se trata de un servicio interactivo de entretenimiento que proporciona al consumidor, a través de su televisor y mando a distancia, un acceso diferente, intuitivo y unificado a sus perfiles en distintas redes sociales. Está formado por dos subsistemas: la interfaz de usuario y los servidores que implementan la lógica del acceso a las redes sociales.



(a) Estructura global de los servicios interactivos



(b) Estructura del servicio de acceso a redes sociales

Figura 3: Arquitectura de los servicios *Syntheractive*

3.1.1. Interfaz del servicio: lo que el ojo ve

Uno de los elementos claves de *Synthsocial* es el elemento que lo define como servicio interactivo y materializa el nexo de unión con la tecnología *Synthetrick* (i.e., clase `Service` en la Figura 4). Este componente se declara como una extensión de la jerarquía de clases para la especificación de aplicaciones para la plataforma *Synthetrick* y funciona como punto de entrada a la aplicación. Implementa el arranque del servicio e incluye las pertinentes llamadas para la inicialización de cualquier otro elemento o subsistema colaborador, así como su respuesta a los distintos eventos y circunstancias que puedan ocurrir durante la ejecución de una sesión de la aplicación: fundamentalmente, el comportamiento anterior y posterior a cada *frame* del contenido visual que se le presenta al consumidor y la respuesta a las pulsaciones de las distintas teclas a las que el servicio debe responder.

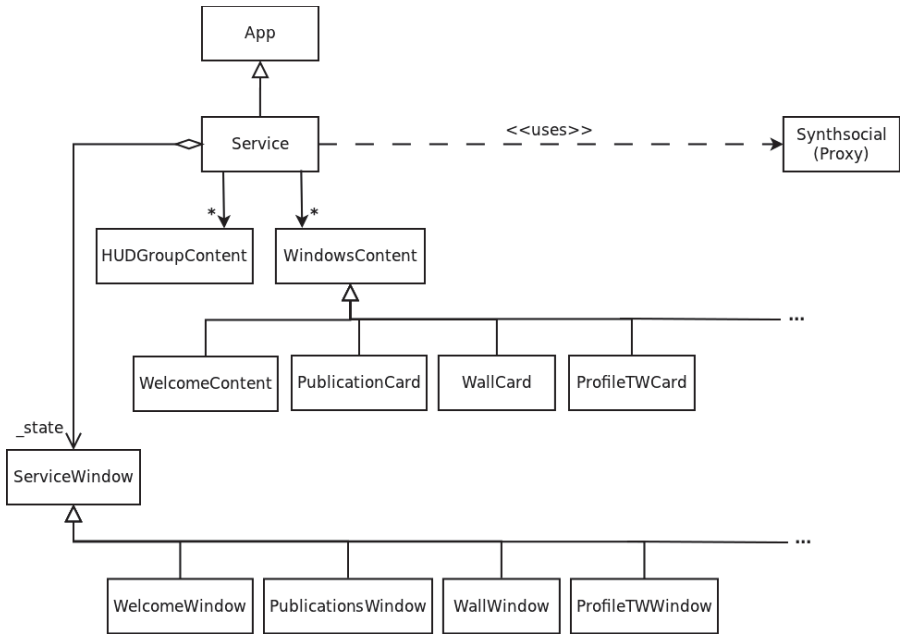


Figura 4: Estructura simplificada del servicio *Synthsocial*

En nuestro caso de estudio hemos decidido separar la definición del comportamiento de respuesta a las interacciones del usuario del resto de la clase principal, mediante el uso del patrón Estado [GHJV95] (i.e., clase `ServiceWindow`). Así, la respuesta del servicio a los distintos eventos, dependiente del estado del propio servicio, se define en subclases diferenciadas (i.e., `WelcomeWindow`, `PublicationsWindow`, etc.), que se corresponden con la pantalla que está visualizando el usuario en cada momento concreto, siendo precisamente la transición de una a otra lo que marca la diferencia en las respuestas ante las mismas interacciones (recuperación de información de una red social, transición de una pantalla a otra, etc.).

Los otros componentes de esta parte del sistema son los que describen los contenidos visuales que se presentan al usuario en cada pantalla (i.e., `WindowsContent`). Por consiguiente, existen también una serie de subclases en las cuales se definen la composición y estructura de estos contenidos para cada una de las pantallas de la aplicación. Todo el contenido visual está desarrollado haciendo uso del motor de gráficos 3D OpenSceneGraph [WQ12], en el que está basado el núcleo del motor de renderizado de la plataforma *Synthetrick*.

OSG conforma una capa de abstracción sobre OpenGL [Shr06] que organiza los contenidos 3D según una estructura jerárquica en árbol conocida como *scene graph*. Así, lo que se debe definir para cada servicio *Syntheractive* como parte de su inicialización gráfica es cuál es el *scene graph* que especifica los componentes visuales del servicio. Cada componente del árbol es un nodo del mismo y existen componentes OSG para representar múltiples tipos de contenido visual: nodos del árbol que permiten agrupar otros nodos bajo ellos, nodos que permiten activar u ocultar todos los que están bajo ellos en el árbol, nodos que permiten cambiar el aspecto y la posición de todos sus hijos, componentes para representar imágenes o texturas, otros para representar información de tipo textual, etc.

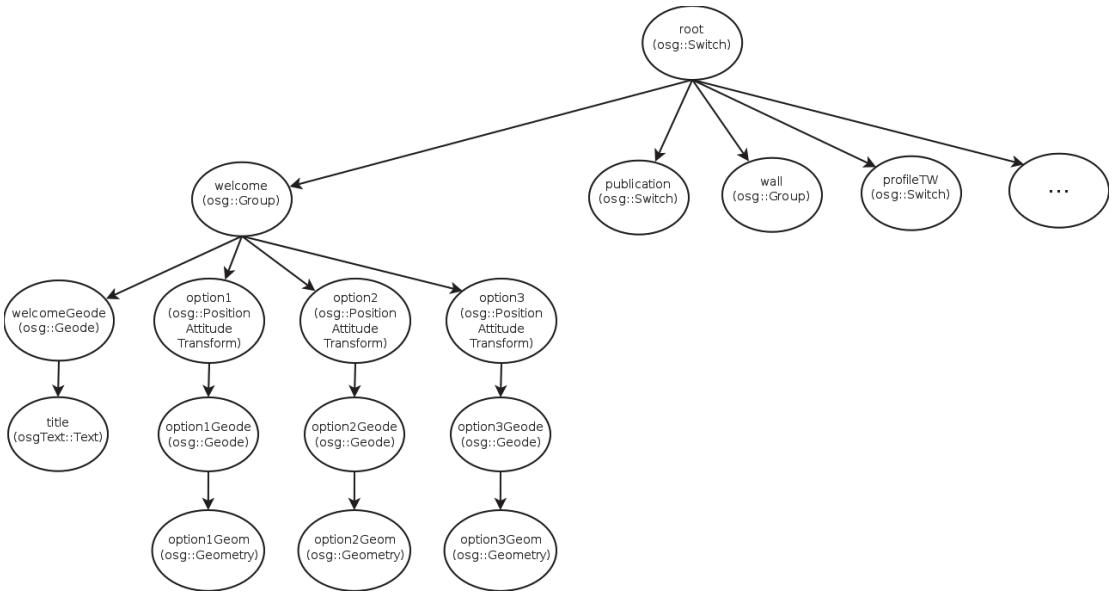


Figura 5: *Scene graph* que define los componentes visuales del servicio *Synthsocial*

El grafo del servicio *Synthsocial* (que se puede ver en la Figura 5, donde el subárbol detallado corresponde a la pantalla principal de bienvenida al usuario) tiene en su raíz un *Switch*, tipo de nodo de OSG que puede tener uno o varios hijos y determina cuáles son visibles y cuáles no en cada momento. Además, existe un hijo de este nodo raíz para cada pantalla o estado posible del servicio, formado por el subárbol correspondiente a dicha pantalla, el cual a su vez define los componentes de ésta y su distribución.

La elección del tipo de nodo para la raíz es importante, ya que utilizar precisamente un *Switch* hace posible manejar de una manera sencilla y eficiente el conjunto completo de subárboles de cada una de las pantallas de la aplicación. De esta manera no es necesario crear y borrar el *scene graph* del servicio cada vez que éste cambia de estado (y, por tanto, se cambian los componentes visuales representados por pantalla), sino que se pueden almacenar en el grafo los nodos con los componentes de todas las pantallas y, mediante la funcionalidad del *Switch*, mostrar sólo aquella parte del árbol que se corresponde con la que se muestra al usuario en cada momento.

3.1.2. Gestión e intermediación de acceso a las redes sociales: el corazón funcional

La lógica de negocio del servicio *Synthsocial*, que gestiona el acceso a la información y la intermediación con las redes sociales, se estructura utilizando *proxies* de acceso a las diferentes redes. Estos *proxies* han sido desarrollados como servidores utilizando el lenguaje de programación Erlang. Forman una estructura de procesos (representada en la Figura 6) en la que uno de los *proxies* ejerce el papel de servidor principal (i.e., `Synthsocial`), punto de entrada común y único para todas las comunicaciones con las sesiones establecidas desde la interfaz del servicio. Por el protocolo definido para tales comunicaciones, como parte de las peticiones que el servicio puede enviar se indica a qué red social se quiere acceder de forma que este servidor

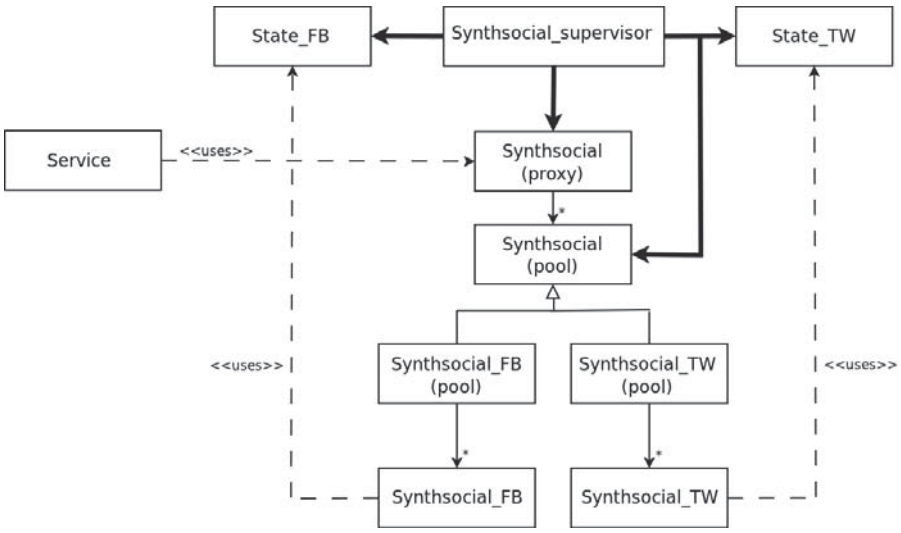


Figura 6: Estructura de servidores y *proxies* de la lógica del servicio

fachada puede encaminar las solicitudes al servidor que corresponda dentro del siguiente nivel de la estructura (i.e., `Synthsocial_FB`, `Synthsocial_TW`, etc.).

El segundo nivel de esta arquitectura está formado, por tanto, por un servidor para cada red social soportada. Cada uno de estos servidores implementa un *pool* de procesos de forma que se posibilita la atención simultánea de múltiples peticiones tanto a la misma como a distintas redes sociales. Los procesos de cada *pool* son los servidores concretos que realmente saben cómo adaptar y manejar las comunicaciones con las APIs de cada una de las redes sociales. Se encargan de enviar a estas APIs las peticiones necesarias para acceder a la información requerida y de transmitir la respuesta de forma adecuada a la sesión del servicio que inició la petición.

Un tercer y último nivel de servidores se encarga del almacenamiento de la información de sesión asociada con cada una de las redes sociales: *tokens* o códigos de acceso para cada uno de los usuarios de la aplicación, caché de ciertos datos recuperados de la red social para su manejo y acceso más eficiente, etc.

Toda esta estructura está organizada en un árbol de supervisión de procesos de forma que un proceso maestro monitoriza todos los servidores mencionados para que, en caso de terminación repentina de alguno de ellos debido a un posible error en su ejecución, sea posible volver a arrancarlo, permitiendo así que todo el subsistema pueda seguir funcionando adecuadamente.

3.1.3. Integración de los subsistemas

Para el correcto funcionamiento de la aplicación es necesario que el subsistema que implementa la lógica pueda recibir las interacciones que realiza el usuario a través de la interfaz de forma que las sesiones del servicio lleguen a canalizar las peticiones a los *proxies* y éstos puedan enviar de vuelta la respuesta resultante de dichas peticiones.

Para posibilitar este intercambio de información se hace uso del módulo de comunicaciones de la plataforma *Synthetrick*, que permite el paso de mensajes entre las sesiones activas de los

distintos servicios que se estén ejecutando en cada momento y entre estas sesiones y las instancias de servidores externos que estén activos dentro del sistema (como sería el caso de la estructura de servidores del *back-end* de *Synthsocial*). Esta comunicación se produce de tal manera que el contenido de estos mensajes es indiferente para el módulo de comunicaciones, que simplemente se encarga de transportarlos a través del canal que se indique desde el origen. El sistema de comunicaciones de *Synthetrick* es el encargado de gestionar esos canales de comunicación.

Cada sesión de un servicio de la plataforma puede enviar mensajes a un determinado canal, o suscribirse para así recibir los que fluyan por él. De esta manera se posibilita la comunicación con el resto de servicios o servidores activos en la plataforma. Además, también puede solicitarse la creación de nuevos canales. Esto es lo que ocurre en el caso del servidor fachada del subsistema de intermediarios de acceso a redes sociales, el cual solicita la creación de un canal propio como parte de su inicialización. De esta forma, las sesiones del servicio *Synthsocial* deben enviar los mensajes con sus solicitudes a dicho canal. A su vez, cada sesión dispone también de un canal propio en el sistema, que se crea al mismo tiempo que la sesión es iniciada. El identificador de este canal debe formar parte de los mensajes que se envían a los servidores de la lógica de negocio. Así, estos sabrán a dónde tienen que enviar sus respuestas para que puedan llegar correctamente a la sesión que inició la solicitud.

Además es necesario un proceso de traducción de los datos entre dos tecnologías diferentes debido a que cada uno de los subsistema está desarrollado con lenguajes distintos (C++ en el caso de la interfaz y Erlang en el de la lógica de negocio). Para solucionar esto se decidió hacer uso de la librería *erl interface* [Eri12] y más concretamente de las funcionalidades de *marshalling* que esta proporciona. Gracias a esto, es posible, desde el código de la interfaz, codificar como un término Erlang las peticiones que se enviarán a los servidores del *back-end*. Así, partiendo de estructuras de datos propias del lenguaje C++ (cadenas de caracteres, *arrays*...), se pueden obtener estructuras que representan cualquier término deseado (en nuestro caso tuplas, *atoms*, listas y *strings*), que será perfectamente interpretable en el lado del receptor. Análogamente, durante la recepción de las respuestas de la lógica de negocio se realiza la operación inversa: es posible descodificar los términos Erlang que llegan en estos mensajes para almacenarlos y manejarlos en estructuras con tipos de datos propios del lenguaje C++.

A modo de ejemplo, la Figura 7 muestra la pantalla principal del servicio *Synthsocial* de bienvenida al usuario, en la que se le da a escoger entre tres opciones: visitar su perfil de Facebook, su perfil de Twitter o consultar en una misma pantalla sus últimas publicaciones en ambas redes sociales. Al lado, se observa la pantalla correspondiente a esta tercera opción, en la que, bajo una apariencia uniforme, se muestran los contenidos provenientes de las dos redes sociales en orden cronológico inverso.

Una vez integrados ambos subsistemas, la aplicación interactiva permite a los consumidores no sólo acceder a los perfiles de sus redes sociales, sino también a efectuar una serie de operaciones de consulta y actualización (visualizar perfiles y publicaciones, escribir mensajes de estado o dirigidos a un usuario, publicar o repetir *tweets*...). La flexibilidad del diseño y estructura presentados permiten además de forma fácil la inclusión de nuevas operaciones para las redes sociales ya soportadas o la incorporación del soporte de acceso a nuevas redes sociales que pudieran considerarse interesantes.

(a) Pantalla de bienvenida (b) Publicaciones en distintas redes sociales

Figura 7: Capturas de pantalla del servicio *Synthsocial* en funcionamiento

3.1.4. Utilización de tecnologías funcionales en *Synthsocial*

Para terminar con esta sección cabe justificar el uso de una tecnología funcional como Erlang para el desarrollo de toda la lógica de negocio de nuestro sistema.

La elección del uso de programación funcional en lugar de algún otro paradigma más tradicional (como la programación estructurada o la orientada a objetos) fue clara una vez analizadas la naturaleza del proyecto y su dominio y contexto de aplicación y las ventajas que aporta este tipo de programación. Asimismo, se escogió el lenguaje Erlang por todas sus características y potencialidades concretas, además de por ser también el lenguaje utilizado de forma subyacente en parte de la tecnología de la plataforma *Synthetrick*.

Este servicio interactivo está destinado a funcionar sobre un sistema distribuido, la plataforma *Synthetrick*. Los nodos sobre los que corren las sesiones de los servicios, los servidores de acceso a redes sociales y el sistema de control que gobierna todo el funcionamiento pueden (e incluso resulta positivo que así sea) residir en GPUs y máquinas diferentes. Además, son necesarias altas capacidades de concurrencia, robustez y disponibilidad para poder dar servicio con una calidad satisfactoria a un gran número de consumidores de forma simultánea. Todo esto supone un punto crítico y lleno de dificultades para su implementación y soporte utilizando tecnologías clásicas de programación. Por contra, utilizando un lenguaje como Erlang, disponemos de una serie de ventajas que facilitan enormemente el proceso de desarrollo, validación y mantenimiento del sistema. Por un lado, la creación y gestión de procesos es trivial y la concurrencia es explícita (sin embargo, en los lenguajes tradicionales el uso de *threads* exige códigos mucho más complejos y propensos a errores). Por otro, Erlang incorpora en su propia sintaxis y semántica el concepto de distribución, por lo que este punto también está resuelto sin ningún tipo de esfuerzo para el desarrollador. Por último, los lenguajes funcionales proporcionan abstracciones mayores que las de los imperativos. Esto provoca que la cantidad de código que es necesario escribir sea menor, que la claridad en la programación de los módulos implicados sea mayor, que se minimicen los efectos colaterales y el número de errores introducidos en el código, etc.

Por consiguiente, *Synthsocial* supone un ejemplo perfecto de todas las ventajas que supone el uso de tecnologías funcionales para un desarrollador de este tipo de sistemas.

4. Discusión

El caso de estudio desarrollado ha permitido diseñar soluciones, flexibles y reutilizables, a diferentes aspectos relacionados con la problemática de la creación de servicios interactivos utilizando la plataforma *Synthetrick*. Por tanto, da la oportunidad de evaluar las ventajas y madurez de esta tecnología de una manera realista y completa.

La utilización del SDK de *Synthetrick*, frente a un enfoque más tradicional que habitualmente hace recaer en el lado del cliente todo el proceso de generación de flujos de vídeo involucrados en un servicio interactivo, presenta toda una serie de ventajas que ya se han mencionado con anterioridad: reducción de costes de implantación y mantenimiento al permitir la utilización de los antiguos STBs ya instalados en los hogares de los clientes; facilidades para desarrollo de aplicaciones como el soporte transparente de múltiples plataformas destino, modos de visualización, comunicaciones inter-sesión, soporte multi-usuario, almacenamiento persistente, etc. Todo esto es posible gracias a la independización, tanto de la interfaz como de la lógica del servicio, con respecto de la combinación hardware/software del cliente final gracias a la generación de flujos de vídeo en el extremo del servidor.

A la hora de desarrollar servicios haciendo uso de esta plataforma, no obstante, algunas consideraciones han de tenerse en cuenta. Uno de los puntos más relevantes es el relacionado con cómo llevar a cabo la implementación del árbol o *scene graph* con los contenidos visuales de cada pantalla de la aplicación. En nuestro caso de estudio, optamos por hacerlo de tal manera que desde el comienzo se almacene la estructura de todas las posibles pantallas y se muestren u oculten en cada momento aquellas que sean convenientes según el estado en que se encuentre el servicio. Una opción alternativa habría sido modificar dinámicamente el *scene graph* asociado al servicio en cada momento de manera que se fueran eliminando del árbol los nodos correspondientes a las pantallas que deben dejar de verse y se fueran generando e introduciendo nuevos nodos con el contenido visual de las nuevas pantallas. Esta alternativa supondría un menor coste de almacenamiento, pues en cada momento sólo tendríamos en memoria los objetos correspondientes a la única pantalla que el usuario está visualizando. Sin embargo, supone una lógica mucho más compleja y propensa a errores. Además, los cambios entre pantallas serían procesados de una manera más lenta debido a la necesidad de modificar todo el grafo desde su raíz y habría que actuar con mucha cautela, ya que realizar un cambio equivocado podría llevar a tener una inconsistencia en el *scene graph* y a obtener resultados inesperados.

La separación, dentro del diseño de la interfaz del servicio, de la lógica de interacción con el usuario y de transición por los distintos estados en una jerarquía y la definición del aspecto visual en otra, unidas gracias a la aplicación del patrón Estado, supone una organización óptima de estos elementos que proporciona al mismo tiempo la flexibilidad necesaria para describir cualquier tipo de interacción, por compleja que resulte, a la vez que se produce una clara separación de responsabilidades que favorece no sólo el reparto de tareas sino también la depuración, la evolución y el mantenimiento.

Otra serie de decisiones importantes han sido las relacionadas con el diseño y la implementación de la lógica de negocio del servicio interactivo, esto es, los servidores intermediarios en el acceso a redes sociales. Una de ellas dio lugar a la inclusión de *pools* de servidores para el acceso a la información de las redes sociales. El uso de la noción de *pool*, en cualquier caso, es clave para soportar el acceso simultáneo al servicio por parte de un gran número de usuarios, algo que

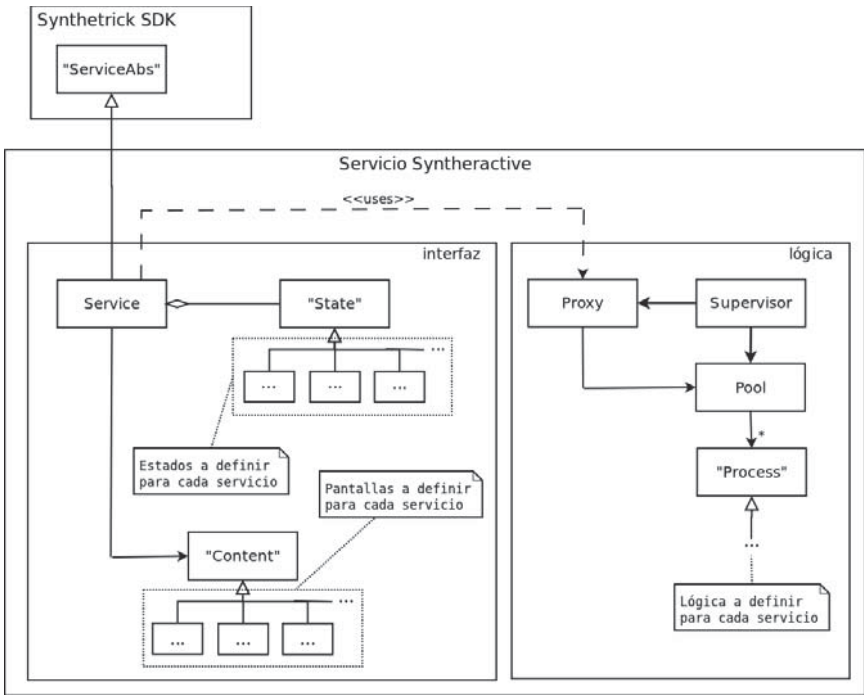


Figura 8: Estructura genérica para un servicio para la plataforma *Synthetrick*

es esperable ya que cada operadora tiene un gran número de clientes que pueden conectarse a través de su STB en el mismo momento. Por otro lado, en este caso no existe el problema del coste computacional asociado como en el caso de la gestión de distintas pantallas, ya que la tecnología utilizada para implementar la lógica del servicio (Erlang) permite trabajar con procesos que son muy ligeros y por tanto muy adecuados para escenarios de alta concurrencia y disponibilidad.

Del mismo modo, el almacenamiento persistente del estado necesario para el funcionamiento de los servidores (*tokens* de acceso por usuario, caché de ciertos datos, etc.) podría llevarse a cabo de diversas maneras. Una de ellas pasaría por utilizar el estado de los servidores *pool* pero se ha optado por separar esta lógica en un nivel adicional de servidores en el cual existe un servidor por cada red social soportada que se encarga de almacenar la información necesaria de dicha red. De esta manera se evita complicar la lógica del servidor *pool* incluyendo funcionalidades que no son propias de este tipo de procesos, manteniéndolo lo más neutral posible y favoreciendo así su reutilización.

4.1. Arquitectura óptima para servicios interactivos

La Figura 8 muestra un diagrama de bloques que representa la estructura genérica recomendada para un servicio *Syntheractive*, basada en el diseño del servicio en base a la separación de la lógica de sus diferentes estados y la descripción gráfica de las distintas pantallas de su interfaz y en la organización de la lógica, externa o interna, en un árbol de supervisión de un *pool* de *proxies*.

Es fácil visualizar cómo cualquier servicio interactivo desarrollado para la plataforma puede adaptarse a esta estructura. Su comportamiento variará a lo largo de la ejecución en función de las interacciones que decida llevar a cabo el usuario, por lo que el servicio transitará dentro de una "máquina de estados" propia. Además, deberá definir un *scene graph* que especifique qué componentes visuales son los que se le presentan al usuario, siendo siempre positivo mantener la implementación de estos contenidos en una jerarquía separada de la que define el comportamiento del servicio en cada uno de sus estados. Por último, para aquellos servicios que necesiten interactuar además con información o elementos externos (como es el caso de nuestro acceso a la información de redes sociales, o sería el de un servicio que necesitara consultar una base de datos o cualquier otro recurso), la mejor opción sería la de disponer de una estructura de servidores *proxies*, supervisada y duplicada gracias a un *pool* de procesos, con la que se comunicará a través del sistema de comunicaciones de *Synthetrick*. De esta manera se proporciona una arquitectura global robusta y con capacidades garantizadas de concurrencia y tolerancia a fallos, todo ello con el mínimo esfuerzo por parte de los desarrolladores.

5. Conclusiones y trabajo futuro

En este artículo presentamos la aplicación interactiva *Synthsocial*, un caso de estudio en el desarrollo de un servicio de televisión interactiva para el acceso a redes sociales utilizando la plataforma *Synthetrick*. Este trabajo nos ha permitido establecer una metodología y arquitectura modelo a seguir por parte de futuros desarrolladores de servicios *Syntheractive*.

La idea que es posible extraer tras la realización de nuestro caso de estudio es que su estructura se puede generalizar y reutilizar en cualquier servicio de la plataforma, llegando incluso a poder empaquetarse y proporcionarse como un *behaviour* o *plantilla* de servicio *Syntheractive*. Esto haría la vida de los desarrolladores de servicios que utilicen la tecnología *Synthetrick* aún más sencilla al permitirles concentrarse únicamente en la implementación de los aspectos específicos de su servicio. Se permitiría así reducir enormemente el ciclo de desarrollo de este tipo de servicios, incluyendo su verificación y validación, lo que lograría además poder integrarlos de manera más sencilla en el catálogo de aplicaciones *Syntheractive*.

En el modelo de arquitectura óptimo propuesto resulta clave la separación de la lógica de presentación y la lógica del servicio y de la lógica de presentación y su definición gráfica. Además, nuestra propuesta incluye el diseño del esqueleto de la lógica del servicio, un esqueleto que proporciona capacidades muy relevantes como el soporte simultáneo de un gran número de peticiones gracias al uso de *pools* de procesos y la robustez y capacidad de recuperación ante problemas de ejecución o errores gracias a la introducción de árboles de supervisión.

Futuras líneas que cabría explorar incluirían la posibilidad de definir otro tipo de servicios que pudiesen requerir arquitecturas alternativas, optimizadas en alguno de sus requisitos (rapidez de ejecución, capacidad de almacenamiento…) que pudieran primar sobre los de concurrencia, robustez y alta disponibilidad aplicados en el caso de *Synthsocial*.

Referencias

[Arm07] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic programmers. Pragmatic Bookshelf, 2007.

[CT09] F. Cesarini, S. Thompson. *Erlang Programming*. O'Reilly Series. O'Reilly, 2009.

[Eri12] Ericsson. Erlang Interface v. 3.7.7. 2012. http://www.erlang.org/documentation/doc-5.9.1/lib/erl_interface-3.7.7/doc/pdf/erl_interface-3.7.7.pdf

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.

[Gol08] J. Goldman. *Facebook Cookbook*. Cookbook Series. O'Reilly Media, Incorporated, 2008.

[Gra08] W. Graham. *Facebook API Developers Guide*. Apress Series. Apress, 2008.

[htt12a] http://www.syntheractive.com/. Syntheractive S.L. 2012.

[htt12b] http://www.syntheractive.com/technology.shtml. Synthetrick SDK. 2012.

[LMC10] M. Logan, E. Merritt, R. Carlsson. *Erlang and OTP in Action*. Manning Pubs Co Series. Manning Publications, 2010.

[Mak09] K. Makice. *Twitter API: Up and Running*. O'Reilly Series. O'Reilly, 2009.

[PGA09] J. Paris, V. Gulías, C. Abalde. A Distributed System for Massive Generation of Synthetic Video Using GPUs. In *Computer Aided Systems Theory - EUROCAST 2009*. Lecture Notes in Computer Science 5717, pp. 239–246. Springer, 2009.

[Shr06] D. Shreiner. *OpenGL programming guide: the official guide to learning OpenGL*. OpenGL Series. Addison-Wesley, 2006.

[SK10] J. Sanders, E. Kandrot. *Cuda by Example: An Introduction to General-Purpose GPU Programming*. Pearson Education, 2010.

[Str09] B. Stroustrup. *Programming: Principles and Practice Using C++*. Developer's Library. Addison-Wesley, 2009.

[TGMR11] J. Taibo, V. Gulías, P. Montero, S. Rivas. GPU-based fast motion estimation for on-the-fly encoding of computer-generated video streams. In *Proceedings of the 21st International Workshop on Network and Operating Systems Support for Digital Audio and Video*. NOSSDAV '11, pp. 75–80. ACM, 2011.

[WQ10] R. Wang, X. Qian. *OpenSceneGraph 3.0: Beginner's Guide*. Packt Publishing, Limited, 2010.

[WQ12] R. Wang, X. Qian. *OpenSceneGraph 3 Cookbook*. Packt Publishing, Limited, 2012.

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): Actas de las "XII Jornadas sobre Programación y Lenguajes (PROLE'2012), Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

Sesión 2

Transformación de programas

Chair: *Dr. Jesús Almendros*

Sesión 2: **Transformación de programas**
Chair: Dr. Jesús Almendros

Jose F. Morales, Rémy Haemmerlé, Manuel Carro and Manuel Hermenegildo. *Lightweight Compilation of (C)LP to JavaScript*.

David Insa, Josep Silva and César Tomás. *Mejora del rendimiento de la depuración declarativa mediante compresión y expansión de bucles*.

Alejandro Acosta, Francisco Almeida and Vicente Blanco. *Paralldroid: a source to source translator for development of native Android applications*.

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): Actas de las "XII Jornadas sobre Programación y Lenguajes (PROLE'2012), Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

Lightweight compilation of (C)LP to JavaScript

Jose F. Morales[1], Rémy Haemmerlé[2], Manuel Carro[1 2], and Manuel V. Hermenegildo[1 2]

IMDEA Software Institute, Madrid (Spain)[1]
School of Computer Science, Technical University of Madrid (UPM), (Spain)[2]
{josef.morales,manuel.carro,manuel.hermenegildo}@imdea.org
remy@clip.dia.fi.upm.es, {mcarro,herme}@fi.upm.es

Abstract: We present and evaluate a compiler from Prolog (extensions) to JavaScript which makes it possible to use (constraint) logic programming to develop the client side of web applications while being compliant with current industry standards. Targeting JavaScript makes (C)LP programs executable in virtually every modern computing device with no additional software requirements from the point of view of the user. In turn, the use of a very high-level language facilitates the development of high-quality, complex software. The compiler is a back end of the Ciao system and supports most of its features, including its module system and extension mechanism. We demonstrate the maturity of the compiler by testing it with complex code such as a CLP(FD) library written in Prolog with attributed variables. Finally, we validate our proposal by measuring the performance of some LP and CLP(FD) benchmarks running on top of major JavaScript engines.

Keywords: Prolog; Ciao; Logic Programming System; Implementation of Prolog; Modules; JavaScript; Web

The Web has evolved from a network of hypertext documents into one of the most widely used OS-neutral environments for running rich applications —the so-called Web-2.0—, where computations are carried both locally at the browser and remotely on a server. Our ambitious objective in [MHCH12] is to enable client-side execution of *full-fledged (C)LP programs* by means of their *compilation* to JavaScript, i.e., to support essentially the full language available on the server side. Our starting point is the Ciao system [HBC+12], which implements a multi-paradigm Prolog dialect with numerous extensions through a sophisticated module system and program expansion facilities. Other approaches often put emphasis on the feasibility of the translation or on performance on small programs, while ours is focused on completeness and integration:

- We share the language front end and implement the translation by redefining the (complex) last compilation phases of an existing system. In return we support a full module system, as well as the existing analysis and source-to-source program transformation tools.
- We provide a minimal but scalable runtime system (including built-ins) and a compilation scheme based on the WAM that can be progressively extended and enhanced as required.
- We offer both high-level and low-level foreign language interfaces with JavaScript to simplify the tasks of writing libraries and integrating with existing code.

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): PROLE'2012, pp. 51-52, ISBN:978-84-15487-27-2.
Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

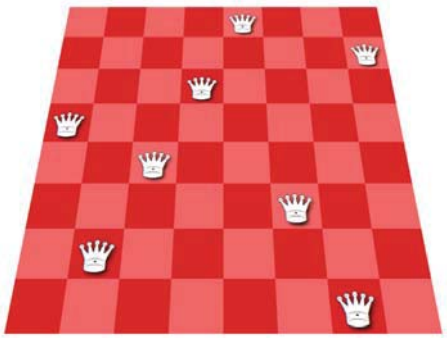


Figure 1: Graphical representation for a solution of Queens-8.

Main Results and Experimental Evaluation. The new back end allows us to read and compile (mostly) unmodified Prolog programs (as well as all the Ciao extensions such as different flavors of (C)LP or functional and higher-order notation, to name a few), run real benchmarks, and, in summary, be able to develop full applications, where interaction with JavaScript or HTML is performed via Prolog libraries and client-side execution in the browser does not require manual recoding. To the extent of our knowledge, ours is the first approach and full implementation which can achieve these goals. Although raw performance is currently not our main goal, we have measured experimentally the performance over a collection of unmodified, small and medium-sized benchmarks, as an initial indication of the size of problems that are amenable to client-side execution with the current implementation. On average, we got a one order of magnitude slowdown w.r.t. the Ciao virtual machine, which is sufficient for a large range of interactive, Web-bound, non computationally-intensive tasks (such as the example Queens solver shown in Fig. 1, fully coded in Prolog). [1]

Conclusions. We believe that our system makes a significant contribution towards the practical feasibility of client-side Web applications based (fully or partially) on (constraint) logic programming, while relying exclusively on Web standards. This reliance makes it possible to execute code on a variety of devices without any need to install additional plug-ins or proprietary code. We believe this is an important advantage, specially since a good number of the currently popular portable devices make such installation hard or impossible. The system is integrated in the Ciao repository and will be included in upcoming Ciao distributions.

Acknowledgements: The research leading to these results has received funding from the Madrid Regional Government under CM project P2009/TIC/1465 (PROMETIDOS), and from the Spanish Ministry of Economy and Competitiveness under project TIN-2008-05624 *DOVES*. The research by Rémy Haemmerlé has also been supported by PICD, the Programme for Attracting Talent / young PHDs of the Montegancedo Campus of International Excellence.

Bibliography

[HBC+12] M. V. Hermenegildo, F. Bueno, M. Carro, P. López, E. Mera, J. Morales, G. Puebla. An Overview of Ciao and its Design Philosophy. *TPLP* 12(1–2):219–252, 2012. http://arxiv.org/abs/1102.5497.

[MHCH12] J. F. Morales, R. Haemmerlé, M. Carro, M. V. Hermenegildo. Lightweight compilation of (C)LP to JavaScript. *Theory and Practice of Logic Programming, 28th Int'l. Conference on Logic Programming (ICLP'12) Special Issue*, 2012. To appear.

[1] This program, as well as more examples, and benchmarks, are publicly available from http://cliplab.org/~jfran/ptojs.

Mejora del rendimiento de la depuración declarativa mediante expansión y compresión de bucles*

David Insa[1], Josep Silva[2], César Tomás[3]

[1] dinsa@dsic.upv.es
[2] jsilva@dsic.upv.es
[3] ctomas@dsic.upv.es
Universitat Politècnica de València
Camino de Vera s/n, E-46022 Valencia, Spain.

Abstract: Uno de los principales objetivos en la depuración es reducir al máximo el tiempo necesario para encontrar los errores. En la depuración declarativa este tiempo depende en gran medida del número de preguntas realizadas al usuario por el depurador y, por tanto, reducir el número de preguntas generadas es un objetivo prioritario. En este trabajo demostramos que transformar los bucles del programa a depurar puede tener una influencia importante sobre el rendimiento del depurador. Concretamente, introducimos dos algoritmos que expanden y comprimen la representación interna utilizada por los depuradores declarativos para representar bucles. El resultado es una serie de transformaciones que pueden realizarse automáticamente antes de que el usuario intervenga en la depuración y que producen una mejora considerable a un coste muy bajo.

Keywords: Depuración declarativa, Árbol de ejecución, *Tree Compression*, *Loop Expansion*

1. Introducción

La depuración es una de las tareas más difíciles y menos automatizadas de la ingeniería del software. Esto se debe al hecho de que los errores están generalmente ocultos tras complejas condiciones que únicamente ocurren en interacciones particulares de componentes software. Normalmente, los programadores no pueden considerar todas las ejecuciones posibles de su software, y precisamente esas ejecuciones no consideradas son las que producen un error. Esta dificultad inherente a la depuración es explicada por Brian Kernighan así:

> *"Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?"*
>
> The Elements of Programming Style, 2nd edition

* Este trabajo ha sido parcialmente financiado por el *Ministerio de Economía y Competitividad (Secretaría de Estado de Investigación, Desarrollo e Innovación)* con referencia TIN2008-06622-C03-02 y por la *Generalitat Valenciana* con referencia PROMETEO/2011/052. David Insa ha sido parcialmente financiado por el Ministerio de Educación con beca FPU AP2010-4415.

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): PROLE'2012, pp. 53-67, ISBN:978-84-15487-27-2.
Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

```
    public class Matrix {
      private int numRows;
      private int numColumns;
      private int[][] matrix;

(1)   public Matrix(int numRows, int numColumns) {
        this.numRows = numRows;
        this.numColumns = numColumns;
        this.matrix = new int[numRows][numColumns];
        for (int i = 0; i < numRows; i++)
          for (int j = 0; j < numColumns; j++)
            this.matrix[i][j] = 1;
      }

(2)   public int position(int numRow, int numColumn) {
        return matrix[numRow][numColumn];
      }
    }
```

```
    public class sumMatrix {
(0)   public static void main(String[] args) {
        int result = 0;
        int numRows = 3;
        int numColumns = 3;
        Matrix m = new Matrix(numRows, numColumns);
        for (int i = 0; i < numRows; i++)
          for (int j = 0; j < numColumns; j++)
            result += m.position(i, j);
        System.out.println(result);
      }
    }
```

Figura 1: Programa iterativo escrito en Java

Los problemas causados por los errores del software tienen un coste muy elevado, muchas veces incluso más que el del propio desarrollo del producto. Por ejemplo, en el informe NIST [NIS02] se calcula que los errores de software no detectados antes de la entrega del producto producen un coste de 59 billones de dólares al año a la economía de EEUU.

A pesar de que se han definido muchas técnicas de depuración automáticas, generalmente se han obtenido pobres resultados. Dos excepciones son la Depuración Delta [Art11, YLCZ11] y la Depuración Declarativa [Sha82, Sil11]. En este artículo presentamos una técnica para mejorar el rendimiento de la Depuración Declarativa reduciendo el tiempo de depuración.

La depuración declarativa es una técnica de depuración semi-automática que genera preguntas sobre los resultados obtenidos en diferentes subcomputaciones. De esta forma, el programador responde a las preguntas y, junto con la información del depurador, es capaz de identificar el error en el código fuente de forma precisa. Hablando en términos generales, el depurador descarta aquellas partes del código fuente que están asociadas a computaciones correctas hasta que se aísla una pequeña parte del código (normalmente una función o procedimiento) donde se encuentra el error. Una propiedad interesante de esta técnica es que el programador no necesita ver el código fuente durante la depuración, únicamente necesita saber el resultado obtenido y el esperado por una ejecución con unas determinadas entradas. De esta forma, si tenemos una especificación formal de los componentes software que es capaz de responder a las preguntas, entonces la técnica es totalmente automática.

Explicaremos la técnica mediante un ejemplo. Considérese el programa Java de la Figura 1. Este programa inicializa los elementos de una matriz a 1 y recorre toda la matriz para sumar todos los elementos. El Árbol de Ejecución (AE) es la versión dinámica del grafo de llamadas y representa con un nodo cada ejecución de un método. El AE asociado a este ejemplo se muestra en la Figura 2 (izquierda). Obsérvese que `main` **(0)** llama al constructor de `Matrix` **(1)**, y entonces llama nueve veces a `position` **(2)** dentro de dos bucles iterativos anidados. La depuración declarativa usa una estrategia de navegación para atravesar el AE preguntando al programador sobre la validez de los nodos. Cada nodo contiene una ejecución diferente de un método con sus entradas y salidas. Teniendo en cuenta esta información, el usuario puede marcar el nodo como

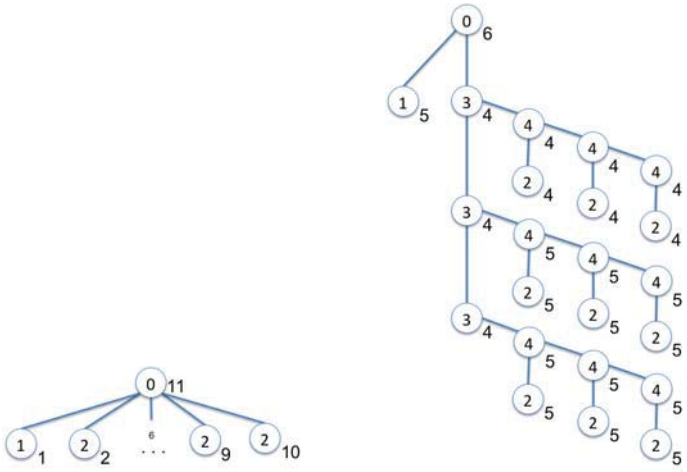


Figura 2: AE de los ejemplos de las Figuras 1 (izquierda) y 8 (derecha)

correcto o *incorrecto*[1] Si el nodo está marcado como incorrecto, entonces el error debe estar en el subárbol de este nodo; y debe estar en el resto del árbol si éste está marcado como correcto. Cuando un nodo es marcado como incorrecto, y todos sus hijos están marcados como correctos, entonces este nodo se convierte en un *nodo erróneo* y el depurador presenta el método asociado como causante del error. En el AE de la Figura 2, cada nodo se ha etiquetado (dentro) con un número que se refiere a la función específica que se ha ejecutado (hemos ignorado los argumentos y resultados por no ser de interés para este trabajo). Además, cada nodo tiene asociado un número (fuera) que indica el número de preguntas necesarias para encontrar el error si este nodo es el nodo erróneo. Para calcular este número, hemos considerado la estrategia de navegación *Divide and Query* (D&Q), que siempre selecciona el nodo que mejor divide el AE por la mitad.

Uno de los principales problemas de la depuración declarativa es que puede producir series de preguntas demasiado largas haciendo que la sesión de depuración se extienda demasiado. Además, la depuración declarativa trabaja a nivel de funciones, de forma que el nivel de granularidad del error encontrado es una función. En este trabajo proponemos una nueva técnica que permite a los depuradores declarativos reducir el número de preguntas necesario para detectar un error, además de permitir reducir el nivel de granularidad a bucles, siendo capaz de reportar un bucle de una función como el causante del error. La técnica se basa en diferentes transformaciones del AE de forma que el AE producido puede ser depurado de forma más eficiente usando los algoritmos estándar. Por lo tanto la técnica es conservativa respecto a las implementaciones anteriores y puede ser integrado en cualquier depurador como una fase de preproceso que sólo afecte al AE. La técnica no sólo reduce el número de preguntas realizadas, sino que puede reducir también la complejidad de las mismas. El coste de las transformaciones es muy bajo comparado con el coste de generar todo el AE y, de hecho, son lo suficientemente eficientes como para usarse siempre en todos los depuradores declarativos antes de la fase de exploración del AE ya que, de acuerdo a nuestros experimentos (véase Tabla 1), como término medio reducen el número de

[1] Existen además otros estados posibles (*desconocido*, *de confianza*, etc.) que, por simplicidad, no consideramos en este artículo por no tener ninguna repercusión sobre las técnicas propuestas [Sil11].

preguntas en un 27.65 %.

El resto del artículo se ha organizado de la siguiente manera. En la sección 2 se revisan otras técnicas y trabajos relacionados. La sección 3 presenta algunas definiciones preliminares que se utilizarán en el resto del trabajo. En la sección 4 explicamos nuestra técnica y sus principales aplicaciones, y se introducen los algoritmos utilizados para mejorar la estructura del AE. La corrección de la técnica se demuestra en la sección 5. En la sección 6 presentamos nuestra implementación y algunos experimentos llevados a cabo con programas Java reales. Finalmente, la sección 7 concluye el artículo. Toda la información relacionada con los experimentos, el código fuente de la herramienta, los casos de prueba y otro material puede encontrarse en http://www.dsic.upv.es/~jsilva/DDJ/.

2. Trabajos Relacionados

Reducir el número de preguntas realizadas por los depuradores declarativos es un objetivo prioritario en el campo, y existen bastantes trabajos orientados a alcanzar este objetivo. Muchos de ellos afrontan el problema definiendo diferentes transformaciones del AE que modifican la estructura del mismo haciendo su exploración más eficiente.

Por ejemplo, en [RVMC11], los autores proponen una técnica para mejorar la depuración declarativa de especificaciones Maude cuyo efecto equilibra el AE. Trabajar sobre un AE equilibrado es conveniente cuando la exploración del AE se realiza con estrategias como *Divide and Query* [Sil11], puesto que después de cada respuesta es posible podar cerca de la mitad del árbol, obteniendo así un número de preguntas logarítmico con respecto al número de nodos. La técnica utiliza una transformación que permite al usuario decidir si quiere mantener los pasos de inferencia de transitividad aplicados por el cálculo. Mantener estos pasos equilibra el árbol y, por lo tanto, se realizan menos preguntas en general. Este enfoque está relacionado con nuestra técnica, pero tiene algunos inconvenientes: sólo puede aplicarse cuando tienen lugar las inferencias de transitividad y, por lo tanto, la mayoría de las partes del árbol no se pueden equilibrar, e incluso en estos casos el balanceo sólo afecta a dos nodos. Nuestra técnica, por el contrario, se aplica a los bucles, siendo capaz de equilibrar todo el bucle.

El enfoque más parecido al nuestro es la técnica *Tree Compression* (TC) introducida por Davie y Chitil [DC06]. Se trata también de un enfoque conservativo que transforma el AE en otro equivalente (más pequeño) en el que se pueden encontrar los mismos errores. El objetivo de esta técnica es esencialmente distinto: intenta reducir el tamaño del AE eliminando nodos redundantes, y sólo es aplicable a llamadas recursivas (sólo trata el caso de la recursión directa). Explicaremos esta técnica mediante varios ejemplos.

Example 1 *Considérese el AE de la Figura 3. Por cada llamada recursiva, TC elimina el nodo hijo asociado a la llamada recursiva y todos los hijos de este nodo pasan a ser hijos del nodo padre. Así, se han eliminado seis nodos de forma que el tamaño del árbol se ha reducido estáticamente. Nótese que el número medio de preguntas (sumatorio del número de preguntas asociadas a cada nodo dividido entre el número total de nodos) ha sido reducido* ($\frac{72}{17}$ *frente a* $\frac{42}{11}$) *gracias a la utilización de TC.*

Desafortunadamente, TC no produce siempre buenos resultados. Algunas veces reducir el

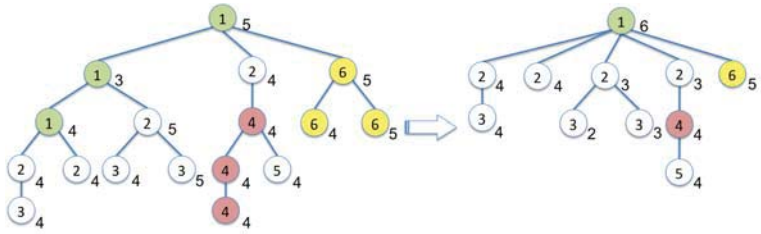


Figura 3: *Tree Compression*

número de nodos causa una estructura del AE que es más difícil de depurar, incrementando así el número de preguntas y produciendo el efecto contrario al deseado.

Example 2 *Considérese el AE de la Figura 4. En este AE, el número medio de preguntas necesarias para encontrar el error es* $\frac{33}{9}$*. Sin embargo, después de comprimir las llamadas recursivas (nodos oscuros), el número medio de preguntas es incrementado a* $\frac{28}{7}$ *(véase el AE de la izquierda). La razón es que el nuevo AE comprimido no puede podar ningún nodo al tener la estructura completamente plana (cualquier nodo marcado como correcto sólo se podará a sí mismo). Nótese que la estructura original nos permite podar algunos nodos ya que los árboles profundos son más adecuados. Sin embargo, si únicamente comprimimos una de las dos llamadas recursivas, el número de preguntas se reduce a* $\frac{27}{8}$*.*

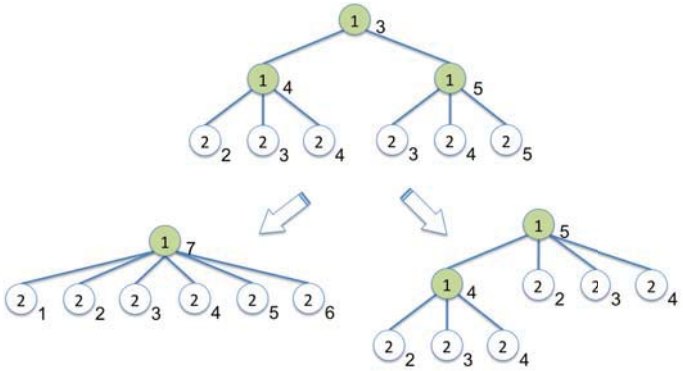


Figura 4: *Tree Compression*

El Ejemplo 2 muestra claramente que TC no debe aplicarse siempre a todas las llamadas recursivas. Sin embargo, las implementaciones existentes siempre comprimen todas ellas [DC06, IS10] puesto que no existe un algoritmo para decidir cuándo aplicar TC y cuándo no. Nuestra técnica soluciona este problema. En particular, una de nuestras transformaciones realiza un análisis para decidir cuándo comprimir llamadas recursivas.

Un enfoque similar a TC es *Declarative Source Debugging* [Cal92] que, en lugar de modificar el árbol, implementa un algoritmo para impedir al depurador seleccionar preguntas relacionadas con los nodos generados por las llamadas recursivas.

Otro enfoque que está relacionado con el nuestro se presentó en [Nil98] donde se introdujo una transformación del código fuente (en lugar del AE) de las listas intensionales de los programas funcionales. Concretamente, esta técnica transforma listas intensionales en un conjunto de funciones equivalentes que implementan la iteración. El AE producido puede ser transformado aún más para eliminar los nodos internos reduciendo el tamaño del AE final como en la técnica TC. Esta técnica es compatible con la nuestra y puede ser aplicada antes de la misma. Es importante destacar que todos los enfoques anteriores fueron definidos en el contexto de los lenguajes funcionales o lógicos. La razón es que la depuración declarativa se definió en el contexto de los lenguajes declarativos y, por lo tanto, gran parte de la investigación se ha llevado a cabo para estos lenguajes. Sin embargo, muchos depuradores declarativos han sido implementados para otros lenguajes como Java (véase [GJ04, CHK06, IS10]), y en estos lenguajes, los bucles se implementan principalmente mediante el uso de iteraciones en lugar de recursión. Por estas razones, muchas de las técnicas anteriores son inútiles para Java, donde las optimizaciones deberían estar orientadas a construcciones iterativas.

Por ejemplo, el AE de la Figura 2 (izquierda) es claramente un AE mal estructurado ya que no permite podar nodos. Fue producido con dos bucles iterativos (anidados) que son una estructura común en los lenguajes imperativos y orientados a objetos. Este tipo de AE es muy común en dichos lenguajes y, por desgracia, no puede ser optimizado con ninguna de las técnicas anteriormente descritas. En las siguientes secciones describimos una técnica capaz de optimizar este tipo de AE.

3. Preliminares

Nuestras transformaciones del AE están basadas en la propia estructura del árbol y en el nombre de la función usada en cada nodo. Esta información es suficiente para identificar todas las cadenas de recursión. Por ello, para el propósito de este trabajo, podemos utilizar una definición de AE en la que los nodos se etiquetan con un número que identifica a una función específica.

Definition 1 (Árbol de Ejecución) Un *Árbol de Ejecución* es un árbol dirigido etiquetado $T = (N, E)$ cuyos nodos N están etiquetados con identificadores de métodos, donde la etiqueta del nodo n es referenciada con $l(n)$. Cada arco $(n \to n') \in E$ indica que el método asociado a $l(n')$ se invoca durante la ejecución del método asociado a $l(n)$.

Usamos números como identificadores de métodos que identifican unívocamente cada método en el código fuente. Esta simplificación es suficiente para mantener nuestras definiciones y algoritmos precisos, y al mismo tiempo simplificarlos. Dado un AE, la *recursión* o *recursión simple* se representa con una rama de nodos enlazados con la misma etiqueta. La *recursión anidada* ocurre cuando una rama que representa una recursión es descendiente de un nodo en otra recursión. La *recursión múltiple* ocurre cuando un nodo etiquetado con un número n tiene dos o más hijos etiquetados con n.

De aquí en adelante, nos referiremos a las dos estrategias de navegación más usadas en la depuración declarativa: *Top-Down* y D&Q. En ambos casos, siempre nos referiremos implícitamente a la versión más eficiente de ambas técnicas, denominadas (i) *Heaviest First* para Top-Down; la cual siempre recorre el AE desde la raíz hasta las hojas seleccionando siempre el nodo con más

descendientes; y (ii) *Hirunkitti's D&Q* que siempre selecciona el nodo en el AE que divide mejor el AE por la mitad. Un estudio comparativo sobre estas técnicas puede encontrarse en [Sil11].

Para la comparación de estrategias usamos la función $Coste(T,s)$ que computa el número necesario de preguntas (como media) para encontrar el error en un AE T usando la estrategia de navegación s. El peso de un nodo representa el número de nodos contenidos en el subárbol cuya raíz es ese nodo. Nos referimos al peso de un nodo n como w_n.

4. Optimización de AE usando *Tree Compression* y *Loop Expansion*

En esta sección presentamos una nueva técnica para la optimización de los AE que combina dos transformaciones independientes: *Tree Compression* (TC) y *Loop Expansion* (LE). TC fue definida y descrita en [DC06]. Aquí introducimos un algoritmo capaz de decidir cuándo comprimir una rama del AE en cualquier caso, incluyendo recursión simple, recursión anidada y recursión múltiple. También discutimos cómo las estrategias se ven afectadas por TC. La otra técnica introducida es LE que, esencialmente, transforma un bucle iterativo en una función recursiva equivalente; y a continuación, agrupa adecuadamente las llamadas recursivas para obtener un nuevo AE tan mejorado como sea posible. Explicaremos cada técnica por separado y mostraremos algoritmos para ellas que pueden trabajar de manera independiente.

4.1. Cuándo aplicar *Tree Compression*

Tree Compression fue propuesta como una técnica general para la depuración declarativa. Sin embargo, fue definida en el contexto de un lenguaje funcional (Haskell) y con el uso de una estrategia particular (Hat Delta). Los propios autores afirmaron que TC puede producir árboles anchos que son difíciles de depurar y, por esta razón, definieron heurísticas que evitaban preguntar sobre la misma función de forma repetida. Desafortunadamente, estas heurísticas sólo son útiles en presencia de llamadas recursivas.

Así, a pesar de que TC puede ser muy útil, también puede producir malos AE como se mostró en la Figura 4. Desafortunadamente, los autores de TC no estudiaron cómo trabaja esta técnica con otras estrategias (más extendidas) como Top-Down o D&Q, de forma que no está claro cuándo usar TC con estas estrategias. Para estudiar esto, consideramos el caso más general de la recursión en un AE como se muestra en la Figura 5, donde las nubes representan conjuntos posiblemente vacíos de subárboles, los nodos oscuros son la cadena de recursión con una longitud de n llamadas, $n \geq 2$, y las etiquetas de los nodos son identificadores.

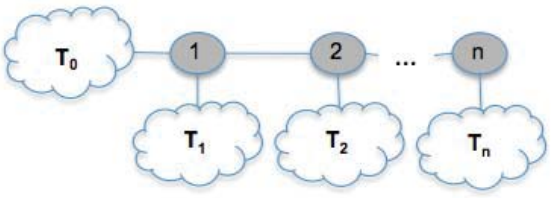


Figura 5: Cadena de recursión en un AE

Debe quedar claro que la cadena de recursión puede ser útil para podar nodos del árbol. Por ejemplo, en la Figura 5, si preguntamos por el nodo n/2, se pueden podar $n/2$ subárboles. En el caso de que los subárboles $T_i, 1 \leq i \leq n$, estén vacíos, entonces no es posible la poda y consecuentemente TC debe ser utilizado. Por lo tanto, podemos concluir que *cada llamada recursiva sin hijos, o cuyo único hijo es otra llamada recursiva, debe ser comprimida*. Este resultado puede ser formalmente establecido para Top-Down de la siguiente manera:

Theorem 1 *Sea T un AE con una cadena de recursión $R = n_1 \rightarrow n_2 \rightarrow \ldots \rightarrow n_m$ donde el único hijo de un nodo n_i, $1 \leq i \leq m-1$, es n_{i+1}. Y sea T' un AE equivalente a T excepto que el nodo n_i ha sido comprimido. Entonces, $Coste(T', Top\text{-}Down) < Coste(T, Top\text{-}Down)$.*

Este teorema ofrece una oportunidad para mejorar de forma estática la estructura de un AE utilizando TC. Sin embargo, TC no es la panacea, y es necesario identificar en qué casos se debe utilizar. D&Q es un buen ejemplo de estrategia en el que TC tiene un efecto negativo.

Tree Compression para D&Q

En general, cuando se depura un AE con la estrategia D&Q, TC no debe ser aplicado, salvo en el caso mostrado por el Teorema 1 ($T_i, 1 \leq i \leq n$, están vacíos). La razón es que D&Q puede saltar a cualquier nodo del AE sin seguir un camino predefinido. Esto permite a D&Q preguntar acerca de cualquier nodo de la cadena de recursividad, sin necesidad de preguntar acerca de los nodos anteriores de la cadena. Nótese que esto no sucede en otras estrategias como Top-Down. Por lo tanto, D&Q tiene la capacidad de utilizar la cadena de recursión como un medio para dividir las iteraciones por la mitad podando nodos.

Dado el AE de la Figura 5, el único caso en el que D&Q no puede utilizar la cadena de recursividad para podar nodos es cuando los conjuntos de subárboles de $T_1 \ldots T_n$ están vacíos. En ese caso, los únicos nodos que pueden podarse son los de la propia cadena y, por consiguiente, es mejor podarlos estáticamente mediante TC. Sin embargo, si los subárboles contienen al menos un nodo, entonces TC *no* debe ser utilizado puesto que la estructura del árbol empeoraría tal y como se muestra en la Figura 6.

Obsérvese en la figura que, a excepción de las cadenas de recursión muy pequeñas (por ejemplo, $n \leq 3$), D&Q puede obtener ventaja de la cadena de recursividad para podar la mitad de las iteraciones. Cuanto mayor es n más nodos se podan. Téngase en cuenta que incluso con un solo hijo en los nodos de la cadena de recursión (por ejemplo, $T_i, 1 \leq i \leq n$ es un único nodo), D&Q puede podar nodos. Por lo tanto, si añadimos más nodos a los subárboles Ti, entonces más nodos pueden ser podados y D&Q se comporta aún mejor. De esta forma podemos concluir que TC no debe ser utilizado en combinación con D&Q excepto en el caso caracterizado en el Teorema 1.

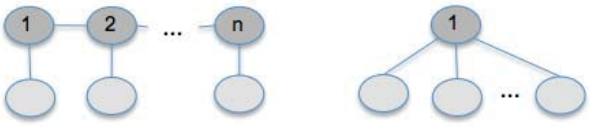


Figura 6: TC aplicado a una secuencia de llamadas recursivas

Tree Compression para Top-Down

En el caso de estrategias basadas en Top-Down, no es trivial en absoluto decidir cuándo aplicar TC y cuándo no aplicarlo. Considerando de nuevo el AE de la Figura 5, hay tres factores que deben tenerse en cuenta para decidir cuándo aplicar TC:

1. la longitud n de la cadena recursiva,
2. el tamaño de cada uno de los árboles $T_i, 1 \leq i \leq n$, y
3. la estrategia de navegación usada.

Con el fin de decidir en qué casos se debe aplicar TC, proporcionamos el Algoritmo 1 que recibe un AE y determina qué nodos de las cadenas de recursión deben ser comprimidos.

En esencia, el Algoritmo 1 analiza para cada posible cadena de recursión cual es el efecto de aplicar TC, y en caso de resultar en una mejora, lo aplica. Esto no se hace con toda la cadena de recursión a la vez, si no que se hace para cada par de nodos recursivos. De tal forma que el resultado final puede ser comprimir solamente una parte (o varias) de la cadena de recursión.

Para ello, la variable *recs* contiene inicialmente todos los nodos del AE que tienen un hijo recursivo. Cada uno de estos nodos se procesa con el bucle de la línea 1 en un proceso desde las hojas hasta la raíz (línea 2). Es decir, primero se procesan aquellos nodos más cercanos a las hojas. Para tratar la recursión múltiple, se usan los bucles de las líneas 5 y 8, que procesan cada rama recursiva de manera independiente y calculan qué ramas deben ser comprimidas para poder obtener una mejora. Dicha mejora se almacena en la variable *improvement*.

El algoritmo hace uso de dos funciones cuyo código ha sido incluido (Cost y Compress), y tres funciones cuyo código no ha sido incluido por ser trivial: la función Children obtiene el conjunto de hijos de un nodo en el AE (i.e., $\text{Children}(m) = \{n \mid (m \rightarrow n) \in E\}$); la función Sort recibe un conjunto de nodos y obtiene una secuencia ordenada donde los nodos han sido ordenados de forma decreciente por sus pesos; y la función Pos recibe un nodo y una secuencia de nodos y devuelve la posición de ese nodo en la secuencia.

Dados dos nodos *parent* y *child* candidatos a ser comprimidos con TC, el algoritmo primero ordena los hijos tanto de *parent* como de *child* (líneas 9-10) en el orden en el que Top-Down preguntaría por ellos (ordenados por su peso). A continuación, se combinan los hijos de ambos nodos para simular una TC (línea 11). Finalmente, se compara el número de preguntas realizadas en ambas opciones (línea 12).

La ecuación de la línea 12 es una de las principales contribuciones de este algoritmo puesto que determina cuándo aplicar TC entre dos nodos de una cadena con la estrategia Top-Down. Esta ecuación depende de la fórmula (línea 21 en la función Cost) usada para calcular el coste medio al aplicar Top-Down.

4.2. Loop Expansion

En general, las llamadas recursivas dividen el AE en diferentes iteraciones representadas con subárboles cuyas raíces pertenecen a la cadena de recursión. Esta estructura es muy conveniente ya que permite la poda de diferentes iteraciones gracias a la cadena de recursión. Por lo tanto, la recursividad es beneficiosa para la depuración declarativa, excepto en los casos expuestos en

Algorithm 1 Tree Compression Optimizado

Entrada: An ET $T = (N, E)$
Salida: An ET T'
Inicialización: $T' = T$ and $recs = \{n \mid n, n' \in N \wedge (n \to n') \in E \wedge l(n) = l(n')\}$

begin
1) **while** $(recs \neq \emptyset)$
2) **take** $n \in recs$ such that $\nexists n' \in recs$ with $(n \to n') \in E^+$
3) $recs = recs \backslash \{n\}$
4) $parent = n$
5) **do**
6) $maxImprovement = 0$
7) $children = \{c \mid (n \to c) \in E \wedge l(n) = l(c)\}$
8) **for each** $child \in children$
9) $pchildren = \text{Sort}(\text{Children}(parent))$
10) $cchildren = \text{Sort}(\text{Children}(child))$
11) $comb = \text{Sort}((pchildren \cup cchildren) \backslash \{child\})$
12) $improvement = \frac{Cost(pchildren)+Cost(cchildren)}{w_{parent}} - \frac{Cost(comb)}{w_{parent}-1}$
13) **if** $(improvement > maxImprovement)$
14) $maxImprovement = improvement$
15) $bestNode = child$
16) **end for each**
17) **if** $(maxImprovement \neq 0)$
18) $T' = \text{Compress}(T', parent, bestNode)$
19) **while** $(maxImprovement \neq 0)$
20) **end while**
end
return T'

function Cost($sequence$)
begin
21) **return** $\sum\{\text{Pos}(node, sequence) * w_{node} \mid node \in sequence\} + |sequence|$
end

function Compress($T = (N, E), parent, child$)
begin
22) $nodes = \text{Children}(child)$
23) $E' = E \backslash \{(child \to n) \in E \mid n \in nodes\}$
24) $E' = E' \cup \{(parent \to n) \mid n \in nodes\}$
25) $N' = N \backslash \{child\}$
end
return $T' = (N', E')$

la sección anterior. Por el contrario, los bucles iterativos producen árboles muy anchos, donde todas las iteraciones son representadas como árboles hijos de una misma raíz. En esta estructura, no es posible podar más de una iteración a la vez, siendo la depuración de estos árboles muy costosa.

Para solucionar este problema, en esta sección se presenta una nueva técnica que transforma los bucles iterativos en funciones recursivas equivalentes. Debido a que la iteración es más eficiente que la recursión, existen muchos enfoques para transformar funciones recursivas en bucles

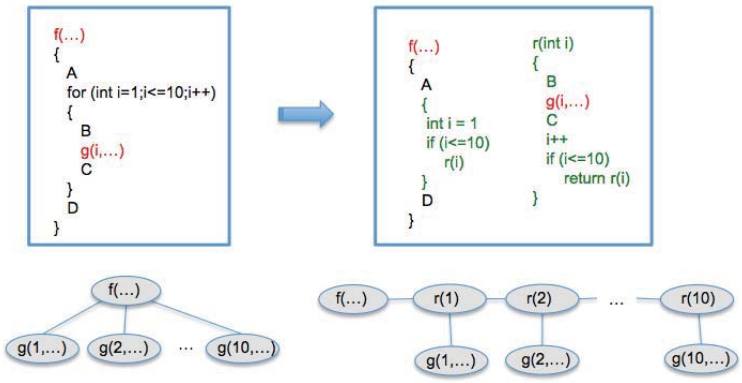


Figura 7: Transformación de un bucle iterativo en una función recursiva equivalente

iterativos (por ejemplo, [HK92, LS00]). Sin embargo, existen muy pocos métodos para transformar bucles iterativos en funciones recursivas equivalentes. Una excepción es la introducida en [YAK00] para mejorar el rendimiento en las jerarquías de memoria multi-nivel. No obstante, no tenemos conocimiento de ningún algoritmo de este tipo propuesto para Java o para cualquier otro lenguaje orientado a objetos. Así, hemos implementado este algoritmo y lo hemos puesto a disposición pública para la comunidad [IS12]. Este algoritmo es la base de LE y básicamente transforma cada bucle iterativo en una función recursiva equivalente. La transformación es ligeramente distinta para cada tipo de bucle (`for`, `foreach`, `while` o `do`).

En el caso de los bucles `for`, se puede explicar con el código de la Figura 7 donde A, B, C y D representan bloques de código. Si observamos el AE transformado vemos que cada iteración se representa con un nodo de la cadena de recursión diferente $r(1) \rightarrow r(2) \rightarrow \ldots \rightarrow r(10)$, de forma que es posible detectar un error en una iteración aislada o en el bucle entero. Esto significa que, en el caso de que la función f tuviera un error, gracias a la transformación, el depurador podría detectar el error en el código B + C o A + D. Nótese que esto no es posible en el AE original, donde el depurador siempre reportaría que A + B + C + D tiene un error. Éste es un resultado interesante puesto que permite aumentar el nivel de granularidad de los errores encontrados, siendo el depurador capaz de detectar los errores dentro de bucles, y no sólo dentro de métodos.

La recursión anidada aumenta las posibilidades de poda. Por ejemplo, la clase Matrix de la Figura 1 puede ser automáticamente transformada[2] al código de la Figura 8. El AE asociado al programa transformado se muestra en la Figura 2 (derecha). Es fácil darse cuenta de que hay una cadena de recursión para cada bucle ejecutado y, por tanto, ahora es posible eliminar bucles, iteraciones, o llamadas individuales dentro de una iteración.

Una vez que LE ha expandido todos los bucles, utilizamos el Algoritmo 1 para volver a comprimir aquellas (sub)cadenas de recursión generadas que no debieron ser expandidas. De esta manera, se consigue tener una representación ideal para cada uno de los bucles del programa.

[2] Por claridad, en la figura hemos cambiado los nombres de las funciones recursivas generadas por `sumRows` y `sumColumns`. En la transformación, si el bucle tiene una etiqueta de bucle (una *label*), se usa esa etiqueta como nombre del bucle. Si la etiqueta no existe, entonces cada bucle tiene como nombre el nombre del método asociado seguido de "_bucleN", donde N es un autonúmerico. Durante la depuración, el depurador muestra al usuario el código del bucle y le permite asociarle otro nombre si éste lo desea.

```
   public class sumMatrix {
(0)   public static void main() {
          int result = 0;
          int numRows = 3;
          int numColumns = 3;
          Matrix m = new Matrix(numRows, numColumns);
          // For loop
          { // Init for loop
             int i = 0;
             // First iteration
             if (i < numRows) {
                Object[] res = sumMatrix.sumRows(m, i, numRows, numColumns, result);
                result = (Integer)res[0];
             }
          }
          System.out.println(result);
       }
(3)    private static Object[] sumRows(Matrix m, int i, int numRows, int numColumns, int result) {
          // For loop
          { // Init for loop
             int j = 0;
             // First iteration
             if (j < numColumns) {
                Object[] res = sumMatrix.sumColumns(m, i, j, numColumns, result);
                result = (Integer)res[0];
             }
          }
          // Update for loop
          i++;
          // Next iteration
          if (i < numRows)
             return sumMatrix.sumRows(m, i, numRows, numColumns, result);
          return new Object[]{result};
       }
(4)    private static Object[] sumColumns(Matrix m, int i, int j, int numColumns, int result) {
          result += m.position(i, j);
          // Update for loop
          j++;
          // Next iteration
          if (j < numColumns)
             return sumMatrix.sumColumns(m, i, j, numColumns, result);
          return new Object[]{result};
       }
   }
```

Figura 8: Versión recursiva del código de la Figura 1

5. Corrección

En esta sección se demuestra que después de nuestras transformaciones, todos los errores que puedan detectarse en el AE original todavía se pueden detectar en el AE transformado. Un resultado aún más interesante es que el AE transformado puede contener más nodos con errores que el AE original. En cuanto a TC, su corrección se ha demostrado en [DC06]. Nuestro algoritmo no influye en esta propiedad de corrección, ya que sólo decide qué nodos deben ser comprimidos y el algoritmo de TC utilizado es el estándar. La completitud y la corrección de LE se formulan a continuación. Las demostraciones asociadas pueden encontrarse en [IST12].

Theorem 2 *[Completitud] Sea P un programa y T su AE asociado, sea P' el programa obtenido al aplicar* Loop Expansion *a P, y sea T' el AE asociado a P'. Para cada nodo erróneo en T, hay al menos un nodo erroneo en T'.*

Theorem 3 *[Corrección] Sea P un programa y T su AE asociado, sea P' el programa obtenido al aplicar* Loop Expansion *a P, y sea T' el AE asociado a P'. Si T' contiene un nodo erróneo asociado al código $f \subseteq P$, entonces T contiene un nodo erróneo asociado al código $g \subseteq P$ y $f \subseteq g$.*

Programa	Nodos				LE	Tiempo		Preguntas				%	
	AE	LE	TC_{ori}	TC_{opt}		LE	TC	AE	LE	TC_{ori}	TC_{opt}	LETC	TC
Factoricer	55	331	51	51	5	151	105	11.62	8.50	7.35	7.35	63.25	100.0
Classifier	25	57	22	24	3	184	4	8.64	6.19	6.46	6.29	72.80	97.36
LegendGame	87	243	87	87	10	259	31	12.81	8.28	11.84	11.84	92.43	100.0
Romanic	121	171	112	113	3	191	12	16.24	7.74	10.75	9.42	58.00	87.62
FibRecursive	5378	6192	98	101	12	251	953	15.64	12.91	9.21	8.00	51.15	86.86
FactTrans	197	212	24	26	3	181	26	10.75	7.88	6.42	5.08	47.26	79.13
BinaryArrays	141	203	100	100	5	172	79	12.17	7.76	7.89	7.89	64.83	100.0
FibFactAna	178	261	44	49	7	202	33	7.90	8.29	8.50	6.06	76.71	71.29
RegresionTest	13	121	15	15	5	237	4	4.77	7.17	4.20	4.20	88.05	100.0
BoubleFibArrays	16	164	10	10	10	213	27	9.31	8.79	4.90	4.90	52.63	100.0
StatsMeanFib	19	50	23	23	6	195	21	7.79	8.12	6.78	6.48	83.18	95.58
Integral	5	8	8	8	3	152	2	6.80	5.75	7.88	5.88	86.47	74.62
TestMath	3	5	3	3	3	195	2	7.67	6.00	9.00	7.67	100.0	85.22
TestMath2	92	2493	13	13	3	211	607	14.70	11.54	15.77	12.77	86.87	80.98
Figures	2	10	10	10	24	597	13	9.00	7.20	6.60	6.60	73.33	100.0
FactCalc	128	179	75	75	3	206	46	8.45	7.60	7.96	7.96	94.20	100.0
SpaceLimits	95	133	98	100	15	786	10	36.26	12.29	18.46	14.04	38.72	76.06

Cuadro 1: Resumen de los experimentos realizados

De los Teoremas 2 y 3 obtenemos un corolario interesante que nos indica que el árbol transformado puede encontrar más errores que el AE original.

Corollary 1 *Sea P un programa y T su AE asociado, sea P' el programa obtenido al aplicar* Loop Expansion *a P, y sea T' el AE asociado a P'. Si T contiene n nodos erróneos, entonces T' contiene n' nodos erróneos con $n \leq n'$.*

6. Implementación

Hemos implementado el algoritmo TC original y el optimizado para transformar árboles de ejecución, y el algoritmo LE de transformación de Java. Todos ellos han sido integrados en el Depurador Declarativo para Java DDJ [IS10, GRS06]. El código fuente de las transformaciones es accesible públicamente en la dirección: http://www.dsic.upv.es/~jsilva/DDJ/.

La implementación de ambos algoritmos se ha hecho en Java. TC tiene aproximadamente 90 líneas de código, y LE alrededor de 1700. Una vez implementadas, hemos realizado varias series de experimentos para medir el impacto que ambas técnicas tienen sobre el rendimiento del depurador. Los resultados obtenidos se resumen en la Tabla 1.

La primera columna de la Tabla 1 indica el nombre del programa depurado. Cada uno de los programas ha sido evaluado asumiendo que el error podría estar en cualquiera de los nodos del árbol de ejecución. Esto significa que cada fila de la tabla es la media de un conjunto de experimentos. Por ejemplo, para evaluar el coste de depurar `Factoricer`, éste se ha depurado 55+331+51+51=488 veces, asumiendo que el error estaba en cada nodo y se ha obtenido la media. Para cada programa, la columna `nodos` muestra el número de nodos que tiene el AE original (`AE`), el AE tras aplicar LE (`LE`), el AE tras aplicar LE primero y después TC en su versión original comprimiendo todos los nodos posibles (TC_{ori}), y el AE tras aplicar LE primero y después TC en su versión optimizada usando el Algoritmo 1 (TC_{opt}); la columna `LE` muestra el número total de bucles expandidos para cada programa; la columna `Tiempo` muestra el tiempo necesario medido en milisegundos para aplicar las transformaciones; la columna `Preguntas` muestra el número medio de preguntas realizadas en cada programa y para cada AE; finalmente, la co-

lumna (%) muestra por un lado el porcentaje de preguntas que se realiza tras aplicar nuestras transformaciones (LE y TC) con respecto al AE original (LETC); y por otro lado el porcentaje de preguntas que se realiza al usar el Algoritmo 1 para decidir cuándo aplicar TC, con respecto a aplicar siempre TC (TC). Como conclusión se observa que utilizar nuestras transformaciones produce una reducción del 27.65 % en el número de preguntas realizadas por el depurador. Por otro lado, el uso del Algoritmo 1 para decidir cuándo aplicar TC y cuando no, también tiene un impacto importante en la reducción del número de preguntas con una media de 9.72 %.

7. Conclusiones

Dos de los principales problemas de la depuración declarativa son (1) que puede generar demasiadas preguntas para encontrar un error, y (2) que una vez que el error se ha encontrado, el depurador reporta un método completo como el código erróneo. En este trabajo hemos dado un paso adelante para resolver estos problemas. Por un lado, hemos introducido técnicas que reducen el número de preguntas mejorando la estructura del AE. Esto se consigue con dos transformaciones llamadas *Tree Compression* y *Loop Expansion*. Por otra parte, *Loop Expansion* también aborda el segundo problema, y nos permite la detección de errores en bucles (y no sólo en métodos) aumentando así el nivel de granularidad del error.

Acknowledgements: Queremos agradecer a Rafael Caballero y Adrián Riesco los comentarios y discusiones productivas realizados en las fases iniciales de este trabajo.

Referencias

[Art11] C. Artho. Iterative Delta Debugging. *International Journal on Software Tools for Technology Transfer (STTT)* 13(3):223–246, 2011.

[Cal92] M. Calejo. *A Framework for Declarative Prolog Debugging*. PhD thesis, New University of Lisbon, 1992.

[CHK06] R. Caballero, C. Hermanns, H. Kuchen. Algorithmic Debugging of Java Programs. In *Proc. of the 2006 Workshop on Functional Logic Programming (WFLP'06)*. Pp. 63–76. Electronic Notes in Theoretical Computer Science, 2006.

[DC06] T. Davie, O. Chitil. Hat-delta: One Right Does Make a Wrong. In *Seventh Symposium on Trends in Functional Programming, TFP 06*. Pp. 27–40. April 2006.

[GJ04] P. Gestwicki, B. Jayaraman. JIVE: Java Interactive Visualization Environment. In *OOPSLA'04: Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. Pp. 226–228. ACM Press, New York, NY, USA, 2004.

[GRS06] F. González-Blanch, F. Roses, S. Serrano. Depurador Declarativo de programas JAVA. In *Master's thesis Universidad Complutense de Madrid*. Available from: http://eprints.ucm.es/9114/, 2006.

[HK92] P. G. Harrison, H. Khoshnevisan. A new approach to recursion removal. *Theor. Comput. Sci.* 93(1):91–113, Feb. 1992.
doi:10.1016/0304-3975(92)90213-Y

[IS10] D. Insa, J. Silva. An Algorithmic Debugger for Java. *In Proc. of the 26th IEEE International Conference on Software Maintenance* 0:1–6, 2010.

[IS12] D. Insa, J. Silva. A Transformation of Iterative Loops into Recursive Loops. Technical report DSIC/05/12, Universitat Politécnica de Valencia, 2012. Available from URL: http://www.dsic.upv.es/~jsilva/research.htm#techs.

[IST12] D. Insa, J. Silva, C. Tomás. Mejora del rendimiento de la depuración declarativa mediante expansión y compresión de bucle. Technical report DSIC/07/12, Universitat Politécnica de Valencia, 2012. Available from URL: http://www.dsic.upv.es/~jsilva/research.htm#techs.

[LS00] Y. A. Liu, S. D. Stoller. From recursion to iteration: what are the optimizations? In *Proceedings of the 2000 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*. PEPM'00, pp. 73–82. ACM, New York, NY, USA, 2000.
doi:10.1145/328690.328700

[Nil98] H. Nilsson. *Declarative Debugging for Lazy Functional Languages*. PhD thesis, Linköping, Sweden, May 1998.

[NIS02] NIST. The Economic Impacts of Inadequate Infrastructure for Software Testing. In Standards and Technology (eds.), *NIST Planning Report 02-3*. May 2002.

[RVMC11] A. Riesco, A. Verdejo, N. Martí-Oliet, R. Caballero. Declarative Debugging of Rewriting Logic Specifications. *Journal of Logic and Algebraic Programming*, September 2011.
doi:http://dx.doi.org/10.1016/j.jlap.2011.06.004

[Sha82] E. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1982.

[Sil11] J. Silva. A Survey on Algorithmic Debugging Strategies. *Advances in Engineering Software* 42(11):976–991, Nov. 2011.
doi:10.1016/j.advengsoft.2011.05.024

[YAK00] Q. Yi, V. Adve, K. Kennedy. Transforming Loops to Recursion for Multi-Level Memory Hierarchies. In *Proceedings of the SIGPLAN 2000 Conference on Programming Language Design and Implementation*. Pp. 169–181. 2000.

[YLCZ11] K. Yu, M. Lin, J. Chen, X. Zhang. Towards Automated Debugging in Software Evolution: Evaluating Delta Debugging on Real Regression Bugs from Developers' Perspectives. *Journal of Systems and Software(JSS)* 85, October 2011.
http://dx.doi.org/10.1016/j.jss.2011.10.016

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): Actas de las "XII Jornadas sobre Programación y Lenguajes (PROLE'2012), Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

Paralldroid: a source to source translator for development of native Android™ applications.

Alejandro Acosta[1] **Francisco Almeida**[1] **and Vicente Blanco**[1]

[1]Dpt. Statistics and Computer Science, La Laguna University, Spain

Abstract: The advent of emergent SoCs and MPSocs opens a new era on the small mobile devices (Smartphones, Tablets, ...) in terms of computing capabilities and applications to be addressed. The efficient use of such devices, including the parallel power, is still a challenge for general purpose programmers due to the very high learning curve demanding very specific knowledge of the devices. While some efforts are currently being made, mainly in the scientific scope, the scenario is still quite far from being the desirable for non-scientific applications where very few applications take advantage of the parallel capabilities of the devices. We propose ParallDroid (Framework for Parallelism in Android™), a parallel development framework oriented to general purpose programmers for standard mobile devices. Paralldroid allows the rapid development of Native Android applications. The user just implements a Java class and introduces Paralldroid annotations. The Paralldroid system automatically generates the C/C ++/OpenCL native code for this class. Paralldroid is provided as a plugin integrated in the eclipse IDE, and it works transparently to the user. The ParallDroid transformation model involves source-to-source transformations and skeletal programming. A proof of concept is presented to test the feasibility, productivity and efficiency of the approach on synthetic applications.

Keywords: SoC, Android, source-to-source transformation

1 Introduction

System on Chip (SoC [SoC12]) has been the enabling technology behind the evolution of many of todays ubiquitous technologies, such as Internet, mobile wireless technology, and high definition television. The information technology age, in turn, has fuelled a global communications revolution. With the rise of communications with mobile devices, more computing power has been put in such systems. The technologies available in desktop computers are now implemented in embedded and mobile devices. We find new processors with multicore architectures and GPUs developed for this market like the Nvidia Tegra [NVIb] with two and five ARM cores and a low power GPU, and the OMAP™4 [Ins] platform from Texas Instruments that Platform also goes in the same direction.

On the other hand, software frameworks have been developed to support the building of software for such devices. The main actors in this software market have their own platform: Android [Goob] from Google, iOS [App] from Apple and Windows phone [Mic] from Microsoft are contenders in the smartphone market. Other companies like Samsung [Sam] and Nokia [Nok] have been developing proprietary frameworks for low profile devices. Coding applications for

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): PROLE'2012, pp. 69-83, ISBN:978-84-15487-27-2.
Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

such devices is now easier. But the main problem is not creating energy-efficient hardware but creating efficient, maintainable programs to run on them [RFGL08].

Conceptually, from the architectural perspective, the model can be viewed as a traditional heterogeneous CPU/GPU architecture where memory performance continues to be outpaced by the ever increasing demands of faster processors, multiprocessor cores and parallel architectures. In particular, embedded memory performance has become the limiting factor in overall system performance for SoC designs. Technologies like Algorithmic Memory [Sys], GPUDirect and UVA (Unified Virtual Addressing) from NVidia [Nvia] and HSA from AMD [Ana] are going in the direction of an unified memory system for CPUs and GPUs.

Under this scenario, we find a strong divorce among traditional mobile software developers and parallel programmers, the first tend to use high level frameworks like Eclipse for the development of Java programs, without any knowledge of parallel programming (Android: Eclipse + Java, Windows: Visual Studio + C#, IOS: XCode + Objective C), and the latter that use to work on Linux, doing their programs directly in OpenCL closer to the metal. The former take the advantage of the high level expressiveness while the latter assume the challenge of high performance programming. The work developed in this paper tries to bring these to worlds.

We propose the ParallDroid system, a development framework that allows for the automatic development of OpenCL parallel applications for mobile devices (Smartphones, Tablets, ...). The developer fills and annotate, using her sequential high level language, the sections on a template that will be executed in parallel. ParallDroid uses the information provided in this template to generate a new parallel program that incorporates the code sections to run over the GPU. The approach does not pose a parallelization of code in the traditional sense (for example through the loop parallelization), nor is a classical parallel skeleton, where the user fills the sequential sections of a parallel skeleton. In our case, starting from the specification given by the programmer, the parallel execution pattern is generated dynamically. ParallDroid can be seen as a proof of concept where we show the benefits of using generation patterns to abstract the developers from the complexity inherent to parallel programs [PAS07].

The advantages of this approach are well known:

- Increased use of the parallel devices by non-expert users
- Rapid inclusion of emerging technology into their systems
- Delivery of new applications due to the rapid development time

We find the novelty of our proposal in the application domain of our framework. Parallelism is often restricted to the scientific domain applications [RS11, BSC], however ParallDroid is oriented to general purpose programmers developping applications, not necessarily scientific, demanding high performance computing. We refer, for example, to applications like those coming from augmented reality, video and image processing, etc.

We chose to focus Paralldroid to Java developers that use Eclipse to develop for the Android that seek to exploit the GPU integrated into the device. Two main reasons lead us to take that decision: Android is an Open Source platform with a very high level of market penetration and it has a large community of developers using Java and Eclipse as the development paradigm.

The paper is structured as follows, in section 2 we introduce the development model in Android and the different alternatives to exploit the devices, some of the difficulties associated to the development model are shown. In section 3 we present the ParallDroid Framework using a synthetic application to illustrate it, the performance of ParallDroid is validated in section 4 using a matrix multiplication problem and a filtering median problem on colour image data. Three different versions have been compared, the Java original version, and the C Native and OpenCL automatic generated versions. The computational results prove the increase of performance provided by ParallDroid at a low cost of development, where the parallelism is hidden to the sequential developers. We finish the paper with some conclusions and future lines of research.

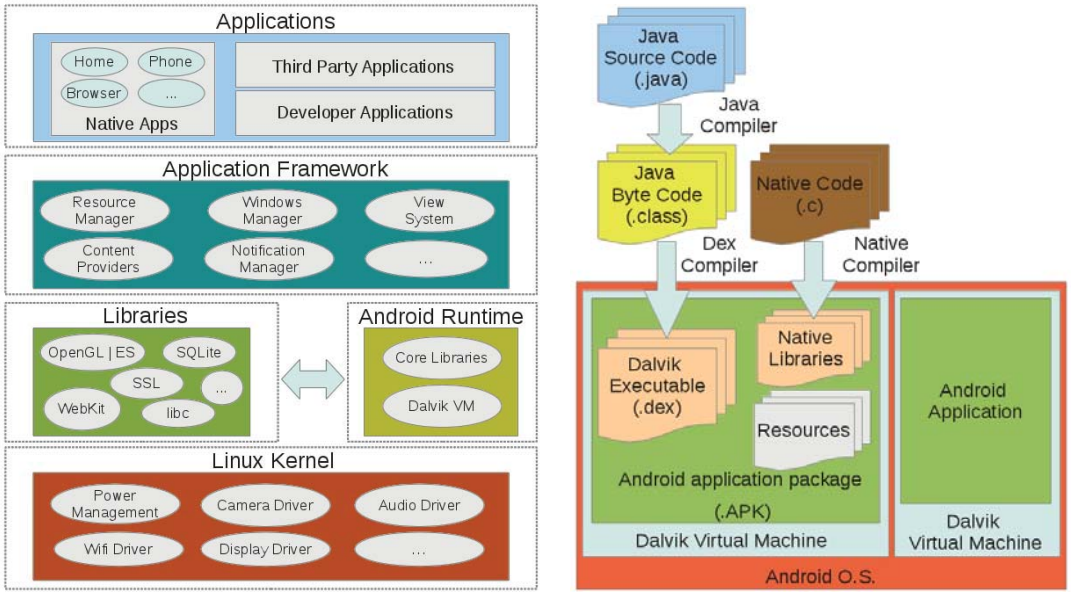
2 The development model in Android

Android™[Gooa] is a software stack for mobile devices that includes an operating system, middleware and key applications (see Figure 1(a)). The Android SDK provides the tools and APIs necessary to begin developing applications on the Android platform using the Java programming language. It has a large community of developers writing applications to extend the functionality of devices.

The SDK tools compile the application code into an Android package (`.apk`) that holds the compiled bytecodes (`.dex`) that will be executed on the Dalvik processing virtual machine (Dalvik VM). The Dalvik VM executes files in the Dalvik Executable (`.dex`) format which is optimized for minimal memory footprint. The VM is register-based, and runs classes compiled by a Java language compiler that have been transformed into the `.dex` format by the included `dx` tool (see Figure 1(b)). Each application lives in its own security sandbox and by default they run in isolation from other applications. Every Android application runs in its own process, with its own instance of the Dalvik VM. Dalvik has been written so that a device can run multiple VMs efficiently. Android relies on Linux for core system services such as security, memory management, process management, network stack, and driver model. The kernel also acts as an abstraction layer between the hardware and the rest of the software stack. The Dalvik VM is supported on the Linux kernel for underlying functionality such as threading and low-level memory management.

Another available development tool is the Android Native Development Kit (NDK). NDK allows to embed components that make use of native code in Android applications. The NDK enables to implement parts of the application running in the Dalvik VM using native-code languages such as C and C++ (see Figure 1(b)). This can provide benefits to certain classes of applications, in the form of reuse of existing code and in some cases increased speed. The Android framework provides several ways to use native code, we use the Java Native Interface (JNI), where developers can implement functions in native code and to load them in the Java code using the function `System.loadLibrary()`. Using native code does not result in an automatic performance increase due to the JNI overload, but always increases application complexity, its use is recommended in CPU-intensive operations that don't allocate much memory, such as signal processing, physics simulation, and so on.

To exploit the high computational capabilities on current devices, the Android SDK provides



(a) Software stack (b) Execution model

Figure 1: The Android software stack and the execution model

RenderScript, it is a high performance 3D graphics rendering computing API at the native level (similar to CUDA) and a programming C language (C99 standard). The main goal is providing the Android developers with a low level API for high performance. During the development process, the C99 code is compiled to an intermediate code that is inserted into the Android package of the application. When the application runs, the intermediate code is compiled and optimized for the device. This makes the code to be portable and improves the performance due to the native code. The RenderScript runtime would decide where the code should be executed (CPU, GPU or any other processing unit available) transparently to the programmer. Currently only the CPU is used and the load is distributed among the CPUs of the system.

The development of applications to exploit the computational capacities in these devices is a complex task. Writing native applications through the NDK requires the knowledge of the native language and the Java Native Interface (JNI) also. RenderScrip partly simplifies the development of applications, however it is still a low level API that developers must learn to use [Gooa]. This is one of the reasons that motivated us to develop ParallDroid.

3 ParallDroid

3.1 The ParallDroid developing model

ParallDroid is designed as a framework to ease the development of future parallel applications on Android platforms. We assume that the mobile platforms will be provided with a classical CPU and with some kind of production processor like a GPU that can be exploited thorough OpenCL. In the proposed translation model, the developers define their problem as a Java class

in the Android SDK. From this class definition, we can generate automatically the OpenCL/C code to be executed in the parallel device.

Just for illustration purposes, we consider the operation of multiplying a vector by a scalar (scalar multiplication). The code in Listing 1 shows a sequential code to solve this operation using the SDK, we can see the loop of lines 12-13 in routine `scalarMultiplication` where the vector is traversed, and the goal is to process this operation in parallel. Following the OpenCL programming model, in ParallDroid each element of the vector can be computed in parallel.

Listing 1: Secuential scalar multiplication.

```
1  public class ScalarMultiplication {
2          public int scalar;
3          public int[]vector;
4
5          public ScalarMultiplication(int size) { // Construct the array
6                  vector = new int[size];
7                  // Initialize array and scalar
8                  // ....
9          }
10
11         public void scalarMultiplication() {
12                 for (int i = 0; i < vector.size(); i++)
13                         vector[i] = vector[i] * scalar;
14         }
15 }
```

The code of Listing 2 shows the specification for the scalar multiplication operation in ParallDroid. The class is annotated as `@Parallel` to denote a parallel class to be processed by ParallDroid. The developer must annotate the routine `scalarMultiplication` to be launched as a Kernel with the directive `@Kernel`. The arguments of a kernel routine have a threefold meaning, the number of arguments indicate the number of dimensions for the execution of the Kernel in the device, the arguments name reference the thread identification when running in parallel, and the value of the parameter represents the number of threads to be executed in this dimension. In this case, since we deal with a one dimensional array, the routine in line 13 receives just one argument and the variable `i` identifies the work to be done by thread `i` in the kernel. Note that, as usual in GPU programming, the loop is removed.

The increment of productivity under this approach is clear, moreover when considering that ParallDroid not only generates the OpenCL code but the C Native JNI implementation. The current version of ParallDroid imposes some constraints that could be overcomed in the future. In the annotated class, we only support primitive type variables, the code inside the kernel must be Java and C99 compatible, more than 3 dimensions are not allowed in the kernel, and depending on the hardware, we may have limitations in the number of threads on each dimension.

3.2 The translation process

Given the advantages provided by IDEs facilitating the development of applications, Paralldroid is designed as an eclipse plugin to generate native code for Android automatically. The developers only have to annotate their classes using a set of annotations and ParallDroid generates the

Listing 2: ParallDroid scalar multiplication developed by end-user.

```
@Parallel
public class ScalarMultiplication {
        public int scalar;
        public int[]vector;

        public ScalarMultiplication(int size) { // Construct the array
                vector = new int[size];
                // Initialize array
                // ....
        }

        @Kernel
        public void scalarMultiplication(int i) {
                vector[i] = vector[i] * scalar;
        }
}
```

native code.

The Paralldroid design is divided into two modules, `the parser` and `the generator`. `The parser` is responsible for parsing of the Java code, identifying annotations and obtaining the data from the annotated Java classes. These data are loaded in the second module, `the generator`. The generator is responsible for generating the native code and modifying the original Java code to use the native code generated.

The parser (Listing 3) is integrated into the Eclipse build process, so when the eclipse build process is started, automatically runs the parser. For parsing Java library class, we use the Java Development Tools (JDT) [Ecl], the library can get all items inside Java class, detecting if a class is annotated, extracting the elds and methods, etc. The parser is responsible for loading the data in the generator class using the information provided by the annotations.

Listing 3: Pseudocode of parser

```
private void parser(...) {
        ...
        GeneratorFactory factory = new GeneratorFactory();
        for (clazz : getAllClass()) {
                classAnnotation = getClassAnnotation(clazz);
                Generator generator = factory.getGenerator(classAnnotation);
                if (generator != null) {
                        getData(generator);
                        ...
                }
        }
}
```

The generator consists of a set of classes (Generator, Field and Method) that dene the methods to be called in the generation process. These classes are responsible for generating the native code and modifying the Java class implemented for the user to invoke the native code generated. In Figure 3 you can see as the process of generation is integrated in the Android execution model (Figure 1(b)).

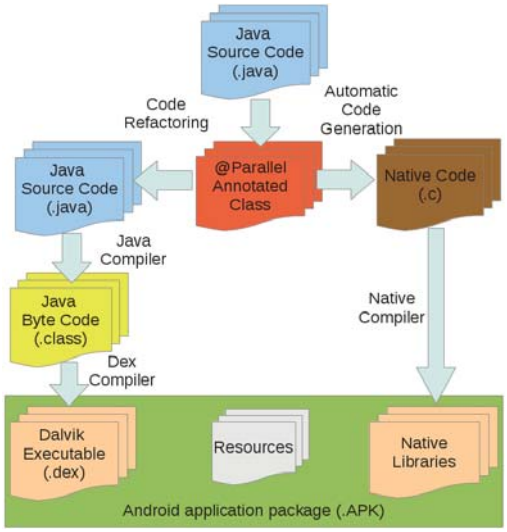


Figure 2: The development model in ParallDroid

When ParallDroid modies the Java class implemented by the user, it adds or changes a set of methods which are described below:

- Inizializer: The Inicializer is responsible for loading the native code generated by ParallDroid using the method System.loadLibrary(), this method is executed when a new instance of the object is created, before the constructor be called.
- Init: The init method allocates the memory in the native environment for all elds dened in the class, a call to this method is added at the end of the class constructor.
- Remove: The remove method releases memory in the native environment for all elds dened in the class, a call to this method is added in the Finalize method.
- Methods annotated with `@Kernel`: In the Java class only is dened the method header, the implementation is dened in the native code generated, to be fully transparent to the user using the same name and parameters dened by the user.

The entire process is transparent to the user, the process for initiating the object is the same as for a Java object, creating an instance and calling to the class constructor. The Java Garbage Collector invokes to Finalize method automatically freeing the memory. Calls to user-dened methods and annotated with `@Kernel` are invoked in the same way and have the same funcionalities than a Java method implementation.

The native code generation process is based in skeletal patterns. In Figure 4 we show the whole translation process for the native class generation and the initialization of references for the `array` variable in the scalar multiplication example. Note how the generic pattern is instantiated with the values corresponding to the instance of generation. There are variable elds in the generic skeletal pattern that are lled with those extracted from the annotated class, in this case, the name

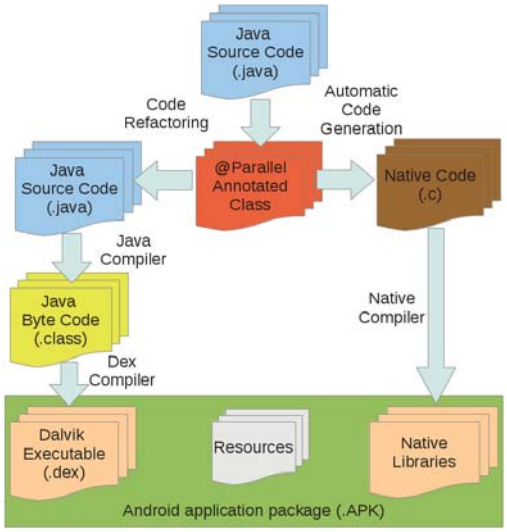


Figure 3: The development model in ParallDroid

of the class (`ScalarMultiplication`), the name of the variable (`vector`) and its type (`int`).

Java Annotated Code

```
@Parallel
public class ScalarMultiplication {
        public int scalar;
        public int[]vector;

        ...
}
```

Skeletal Generation Pattern

```
$className = "ScalarMultiplication";
$arrayName = "vector";
$arrayType = "int";

$generateCode = "
    j$(arrayType)Array GEN$(arrayName);

    JNIEXPORT void JNICALL Java_$(className)_init(JNIEnv *env,
                                                  jobject obj,
                                                  j$(arrayType)Array $(arrayName)) {
        GEN$(arrayName) = (j$(arrayType)Array)(*env).NewGlobalRef($(arrayName));
        (*env).DeleteLocalRef($(arrayName));
    }
"
```

Native Generated Code

```
jintArray GENvector;

JNIEXPORT void JNICALL Java_ScalarMultiplication_init(JNIEnv *env,
                                                      jobject obj,
                                                      jintArray vector) {
   GENvector = (jintArray)(*env).NewGlobalRef(vector);
   (*env).DeleteLocalRef(vector);
}
```

Figure 4: The generation process from skeletal generation pattern

For the scalar multiplication example, ParallDroid modies the Java code implemented by the user(Listing 2) to get the code that is shown in Listing 4. In lines 24-25 you can see the code used by the user for initialize and invoke the native methods, note that it is the same process that would be used with a Java implementation.

Listing 4: Java code generated by ParallDroid.

```
1  public class ScalarMultiplication {
2          public int scalar;
3          public int[]vector;
4
5          {       // Initializer method, runs before any other constructor
6                  System.loadLibrary("ScalarMultiplication");
7          }
8
9          public ScalarMultiplication(int size) { // Construct the array
10                 vector = new int[size];
11                 // Initialize array ...
12                 init(vector);                  // Generated automatically
13         }
14
15         // Code generated for the native execution
16         public native void scalarMultiplication(int i);
17         public native void init(int[] vector);
18         public native void remove();
19         protected void finalize () throws Throwable {remove();}
20 }
21 // Executing the native code
22 int vectorSize = ...;
23 int numThreads = vectorSize;
24 ScalarMultiplication sm = new ScalarMultiplication(vectorSize);
25 sm.ScalarMultiplication(numThreads); // Launching numThreads Threads
```

In the Listing 5 you can see the native code generated of the method annotated with `@kernel` (Listing 2). In line 3 shows the denition of the kernel. In line 10 shows the native implementation of the method, rst we get the current values of elds in the Java class (lines 12-16), next, initialize the variable OpenCL (lines 18-20), allocate memory and loads the data on the GPU (lines 22-25), next, we invoke the kernel (lines 27-38), we get the results (lines 40-43), we release the OpenCL memory (line 45) and put the results in the Fields in Java class (lines 47-48).

4 Computational Results

To validate the performance of the code generated by our framework, we consider two different applications, a classical synthetic matrix multiplication, and a median filtering of an image in the spatial domain [AK97]. In both cases, we implemented three versions of code, the ad-hoc version from a Java developer, ad-hoc C native implementation, and the native OpenCL code automatically generated by ParallDroid. The ParallDroid code generation is hundred percent functional in the Android Emulator, however, the OpenCL libraries are still not available for Android. For that reason, for the validation of the performance, the running times of the three versions have been measured on a system composed of a CPU Intel i3 2130 with 4 cores and a GPU Nvidia GT440.

Listing 5: OpenCL/C code generated by ParallDroid.

```
jintArray GENvector;
// The Kernel
const char * scalarMultiplication = "__kernel void scalarMultiplication("
                                    "int scalar, __global int* vector){"
                                    "int i = get_global_id(0);"
                                    "  vector[i]=vector[i] * scalar;"
                                    "}";

// The public native void scalarMultiplication(int i);
JNIEXPORT void JNICALL Java_ScalarMultiplication_scalarMultiplication(
                                  JNIEnv *env, jobject obj, jint i)  {
        jclass c = (*env).GetObjectClass(obj);
        // JNI to C data transformation
        jfieldID IDscalar = (*env).GetFieldID(c, "scalar", "I");
        jint scalar = (*env).GetIntField(obj, IDscalar);
        jint *vector = (*env).GetIntArrayElements(GENvector, 0);
        // Initialize OpenCL specific variables
        ...
        // OpenCL device memory for arrays
        cl_mem d_vector;
        // Initialize OpenCL: GPU devices, command-queue, device memory
        d_vector = clCreateBuffer(clGPUContext,
                        CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
                        (*env).GetArrayLength(GENvector)*sizeof(int),
                        vector, &errcode);
        // Load and build OpenCL kernel
        kernelLength = strlen(scalarMultiplication);
        clProgram = clCreateProgramWithSource(clGPUContext, 1,
                       (const char **)&scalarMultiplication,
                       &kernelLength, &errcode);
        clBuildProgram(clProgram, 1, clDevices, NULL, NULL, NULL);
        clKernel=clCreateKernel(clProgram,"scalarMultiplication",&errcode);
        // Launch OpenCL kernel
        clSetKernelArg(clKernel, 0, sizeof(int), (void *)&scalar);
        clSetKernelArg(clKernel, 1, sizeof(cl_mem), (void *)&d_vector);
        globalWorkSize[0] = i;
        clEnqueueNDRangeKernel(clCommandQue,clKernel,1,NULL,
                               globalWorkSize,NULL,0,NULL,NULL);
        // Retrieve result from device
        clEnqueueReadBuffer(clCommandQue,d_vector,CL_TRUE,0,
                            (*env).GetArrayLength(GENvector)*sizeof(int),
                            vector,0,NULL,NULL);
        ...
        // Free OpenCL device memory for arrays
        clReleaseMemObject(d_vector);
        // C to JNI data Transformation
        (*env).SetIntField(obj, IDscalar, scalar);
        (*env).ReleaseIntArrayElements(GENvector, vector, 0);
}
```

Without loss of generality we hope that the computational results obtained here will be extrapolated to SoCs supporting Android and OpenCL since the virtual machine, the libraries and architectures will be analogous. To emulate the conditions of the Android system the running times have been measured under a Java Virtual machine. Times are expressed in milliseconds. The input/output data times have not been included in these running times, but the transference

Size	Execution					
	Java		C Native		OpenCL	
	Time	Ratio	Time	Ratio	Time	Ratio
500x500	262	1x	160	1.6x	673	0.4x
1000x1000	6958	1x	6591	1.1x	5099	1.4x
2000x2000	58152	1x	57142	1x	27747	2.1x

Table 1: Performance of the Matrix Multiplication

times among the different memory systems are included, i. e., the data movement between the Java Virtual Machine and the main memory for the native code, and the time between the main memory and the memory of the GPU. Labels Java, C Native and OpenCL show, in tables and figures, the running time and the ratio with the Java sequential version. Since the two problems are two-dimensional, the execution of the OpenCL implementation launches sequentially kernels on one of the dimensions, i.e., for matrices of size n, we launch n times the kernel using n threads each. Other combinations of kernel/thread may produce different running times.

The annotated code for the matrix product can be seen in Listing 6, it implements a two-dimensional kernel where each thread is intended to calculate the value of an element `C[i][j]` from the resulting matrix. Square matrices of sizes 500, 1000, 2000 have been tested. Table 1 and Figures 5(a) show the time in milliseconds and the performance obtained. We observe that for the small problem the C Native version performs better with a ratio of *1.6x* but when the size of the problem increases the OpenCL implementation is better. A ratio of *2.1x* is obtained. This ratio seems to increase on the OpenCL code when the size of the problem is incremented.

Listing 6: ParallDroid Kernel for a Matrix Multiplication

```
1  @Parallel
2  public class MatrixMultiplication {
3     private int size;
4     private int[] A, int[] B, int[] C;
5     ...
6
7     @Kernel
8     public void MatrixMult(int i, int j) {
9        int value = 0;
10       for (int k = 0; k < size; ++k) {
11          int elementA = A[j * size + k];
12          int elementB = B[k * size + i];
13          value += elementA * elementB;
14       }
15       C[j * size + i] = value;
16    }
17 }
```

The code in Listing 7 shows the ParallDroid kernel for the median filtering problem considering colour image data. Again it is a two-dimensional kernel, for each pixel (x,y), the filtering is applied by replacing each entry with the median of the neighbouring entries (*window*) on each dimension RGB. Image data of sizes 1024×768, 3872×2592 and windows of size 3×3, 6×6,

12×12, 24×24 have been tested. In this case, the difficulty of the problem is incremented with the size of the problem and with the size of the window. Table 2 and Figures 5(b) show the time in milliseconds and the performance. We observe the same behaviour than in the former problems for small problems the C Native version is the best, and when the difficulty of the problem increases, the OpenCL implementation overcomes them. In this case, since we are dealing with larger sized problems the performance is incremented reaching a value of *5.4x* for the improvement ratio.

This computational experience proved that ParallDroid offers good performances with a very low cost of implementation where the parallelism is hidden to the sequential developer. As expected, in most of the cases, the C Native implementation improves to the Java implementation so, ParallDroid is an useful tool also for the automatic generation of the native code. As usual, to obtain a substantial gain from the parallel implementation the size of the problem must be large enough. At this point a comparative with RenderScript would be mandatory. However since RenderScript compiles the native code in running time a fair comparative between both environment will not be fair until the API Android/OpenCL be provided.

Listing 7: ParallDroid Kernel for a Filtering Problem

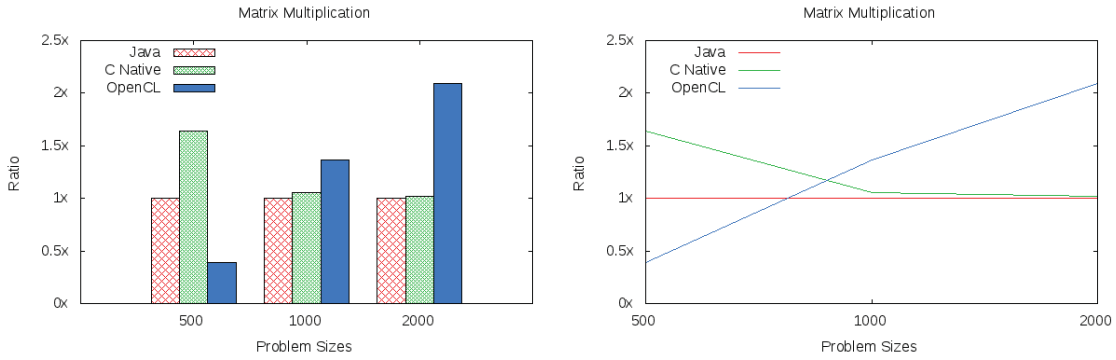
```
1  @Parallel
2  public class MedianFilter {
3     private int window, height, width;
4     private int r[], g[], b[];
5     ...
6
7     @Kernel
8     public void filter(int x, int y) {
9        int disp;
10       int valueR = valueG = valueB = 0;
11       for (int i = 0; i < window; i++) {
12          disp = ((x+i)*height);
13          for (int j = 0; j < window; j++) {
14             valueR += r[disp+(y+j)];
15             valueG += g[disp+(y+j)];
16             valueB += b[disp+(y+j)];
17          }
18       }
19       r[((x)*height)+y] = valueR/(window*window);
20       g[((x)*height)+y] = valueG/(window*window);
21       b[((x)*height)+y] = valueB/(window*window);
22    }
23 }
```

5 Conclusion and Future Work

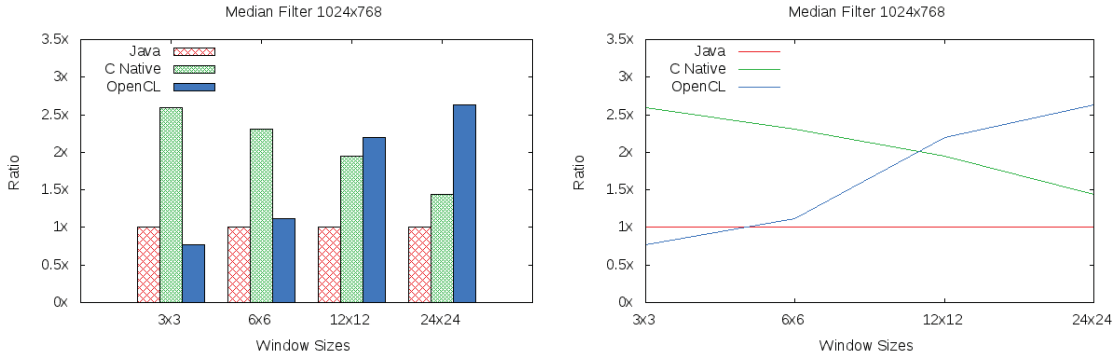
We propose ParallDroid, a framework to generate parallel OpenCL applications for Android. The Java code annotated by the user is automatically transformed in a native parallel version. The generation process is automatic and transparent for the Java developer that has no knowledge on parallel programming. Although there is still opportunity for the optimization in terms of the memory transfer among the different devices, the validation test performed on two different problems prove that the results are quite promising. With a very low development effort the running

Size	Execution	Window Size							
		3x3		6x6		12x12		24x24	
		Time	Ratio	Time	Ratio	Time	Ratio	Time	Ratio
1024x768	Java	57	1x	97	1x	255	1x	680	1x
	C Native	22	2.6x	42	2.3x	131	1.9x	472	1.4x
	OpenCL	74	0.8x	87	1.1x	116	1.7x	258	2.6x
3872x2592	Java	607	1x	1137	1x	3220	1x	8333	1x
	C Native	275	2.2x	542	2.1x	1554	2.1x	5212	1.6x
	OpenCL	353	1.7x	401	2.8x	601	5.4x	1585	5.3x

Table 2: Performance of the Filtering Median Problem



(a) Matrix Multiplication



(b) Filtering Median Problem, 1024x768

Figure 5: Performance of tested problems

times are significantly reduced. ParallDroid also contributes to increase the productivity in the parallel developments due to the low effort required. For the near future we plan to introduce further optimizations in the memory transfers. We will also focus now using ParallDroid for the parallelization of basic libraries used for Android programmers that could take advantage of the parallel execution.

Acknowledgements: This work has been supported by the EC (FEDER) and the Spanish MEC with the I+D+I contract number: TIN2008-06570-C04-03 and TIN2011-24598

Bibliography

[AK97] J. Astola, P. Kuosmanen. *Fundamentals of nonlinear digital filtering*. Electronic engineering systems series. CRC Press, 1997.
http://books.google.es/books?id=4foUVdUxSp0C

[Ana] Anandtech. AMD Outlines HSA Roadmap: Unified Memory for CPU/GPU in 2013, HSA GPUs in 2014. http://www.anandtech.com/show/5493/amd-outlines-hsa-roadmap-unified-memory-for-cpugpu-in-2013-hsa-gpus-in-2014.

[App] Apple. iOS: Apple mobile operating system. http://www.apple.com/ios.
http://www.apple.com/ios

[BSC] BSC. MontBlanc Project: High Peformance Computing with embedded and mobile devices. http://www.montblanc-project.eu/.
http://www.montblanc-project.eu/

[Ecl] Eclipse. Eclipse JDT. http://www.eclipse.org/jdt/.
http://www.eclipse.org/jdt/

[Gooa] Google. Android Developers. http://developer.android.com/index.html.
http://developer.android.com/index.html

[Goob] Google. Android mobile platform. http://www.android.com.
http://www.android.com

[Ins] T. Instruments. OMAP^{TM}Mobile Processors : OMAP^{TM}4 Platform. http://www.ti.com/omap4.
http://www.ti.com/omap4

[Mic] Microsoft. Windows Phone: Microsoft mobile operating system. http://www.microsoft.com/windowsphone.
http://www.microsoft.com/windowsphone

[Nok] Nokia. Nokia Belle: lastest Nokia symbian platform. http://www.developer.nokia.com/.
http://www.developer.nokia.com/

[Nvia] Nvidia. GPUDirect Technology. http://developer.nvidia.com/gpudirect.
http://developer.nvidia.com/gpudirect

[NVIb] NVIDIA. NVIDIA Tegra mobile processors: Tegra2 and Tegra 3. http://www.nvidia.com/object/tegra-superchip.html.
http://www.nvidia.com/object/tegra-superchip.html

[PAS07] I. Peláez, F. Almeida, F. Suárez. DPSKEL: A skeleton based tool for parallel dynamic programming. In *Seventh International Conference on Parallel Processing and Applied Mathematics, PPAM2007*. 2007.

[RFGL08] A. D. Reid, K. Flautner, E. Grimley-Evans, Y. Lin. SoC-C: efficient programming abstractions for heterogeneous multicore systems on chip. In Altman (ed.), *Proceedings of the 2008 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES'08*. Pp. 95–104. ACM, Atlanta, GA, USA, Oct. 2008.

[RS11] R. Reyes, F. de Sande. Automatic code generation for GPUs in llc. *The Journal of Supercomputing* 58(3):349–356, 2011.

[Sam] Samsung. Bada: Samsung mobile operating system. http://developer.bada.com. http://developer.bada.com

[SoC12] SoCC. IEEE International System–on–Chip Conference. http://www.ieee-socc.org/, Sept. 2012. http://www.ieee-socc.org/

[Sys] M. Systems. Algorithmic Memory ^{TM}technology. http://www.memoir-systems.com/. http://www.memoir-systems.com/

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): Actas de las "XII Jornadas sobre Programación y Lenguajes (PROLE'2012), Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

Sesión 3

Bases de Datos

Chair: *Dr. Ginés Moreno*

Sesión 3: Bases de Datos
Chair: Dr. Ginés Moreno

Fernando Saenz-Perez. *Tabling with Support for Relational Features in a Deductive Database*.

Rafael Caballero, Jose Luzon-Martin and Antonio Tenorio. *Test-Case Generation for SQL Nested Queries with Existential Conditions*.

Jesús M. Almendros-Jiménez, Alejandro Luna Tedesqui and Ginés Moreno. *Debugging Fuzzy-XPath Queries*.

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): Actas de las "XII Jornadas sobre Programación y Lenguajes (PROLE'2012), Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

Tabling with Support for Relational Features in a Deductive Database

Fernando Sáenz-Pérez[1]*

Grupo de programación declarativa (GPD),
Dept. Ingeniería del Software e Inteligencia Artificial,
Universidad Complutense de Madrid, Spain[1]

Abstract: Tabling has been acknowledged as a useful technique in the logic programming arena for enhancing both performance and declarative properties of programs. As well, deductive database implementations benefit from this technique for implementing query solving engines. In this paper, we show how unusual operations in deductive systems can be integrated with tabling. Such operations come from relational database systems in the form of null-related (outer) joins, duplicate support and duplicate elimination. The proposal has been implemented as a proof of concept rather than an efficient system in the Datalog Educational System (DES) using Prolog as a development language and its dynamic database.

Keywords: Tabling, Outer Joins, Duplicates, Relational databases, Deductive databases, DES

1 Introduction

Tabling is a useful implementation technique embodied in several current logic programming systems, such as B-Prolog [ZS03], Ciao [GCH+08], Mercury [SS06], XSB [SW10], and Yap Prolog [RSC05], to name just a few. This technique walks by two orthogonal axes: performance and declarative properties of programs. Tabling enhances the former because repeated computations are avoided since previous results are stored and reused. The latter axis is improved because order of goals and clauses are not relevant for termination purposes. In fact, tabled computations in the context of finite predicates and bounded term depths are terminating, a property which is not ensured in top-down SLD computations.

Deductive database implementations with Datalog as query language have benefited from tabling [RU93, SSW94a, SP11] as an appropriate technique providing performance and a framework to implement query meaning. Terminating queries is a common requirement for database users. Also, the set oriented answer approach of a tabled system is preferred to the SLD one answer at a time.

However, "relational" database systems embed features which are not usually present altogether in deductive systems. These include duplicates, which were introduced to account for bags of data (multisets) instead of sets. Also, the need for representing absent or unknown information delivered the introduction of null values and outer join operators ranging over such

* This author has been partially supported by the Spanish projects STAMP (TIN2008-06622-C03-01), Prometidos-CM (S2009TIC-1465) and GPD (UCM-BSCH-GR35/10-A-910502)

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): PROLE'2012, pp. 87-101, ISBN:978-84-15487-27-2.
Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

values. Finally, aggregate functions allow to compute summarized data in terms of (multi)sets and considering null occurrences. So, these systems are not relational anymore as they depart from the original model [Cod70], where these features are not considered. In addition, the introduction by major vendors of some of these features in databases are claimed as error sources and unnecessary [Dat09], but it is a fact that they are widely used by database practitioners. However, this is rather a debate out of the scope of this paper.

Thus, the aim of this paper is to show how such features can be supported altogether in a tabled deductive system with Datalog as a query language. To this end, we base our presentation on the grounds of DES (Datalog Educational System) [SP11], a system implemented in Prolog which runs on different platforms (both Prolog and OS's), which allows to easily test new engine implementations. This work extends [SP11] by describing how tabling is implemented and introducing (tabled) duplicates. Supported Prolog platforms along time include Ciao, GNU Prolog, SICStus Prolog and SWI-Prolog. Because it was thought to be as platform independent as possible, tabling in particular was implemented as no supported platform provided it (Ciao recently added this feature). Also, proprietary systems as SICStus Prolog do not provide open sources to modify parts of the system, as it would be the case for implementing tabling.

The very first motivation for including such features in this system came for the need to support SQL as a query language in a deductive database. In DES, both Datalog and SQL are supported, and SQL statements are compiled to Datalog programs and eventually solved by the deductive inference engine. So, embodying nulls, outer joins and duplicates became a need. However, as the system was intended for educational purposes since its inception, it is not targeted at performance and also lacks features such as concurrency, security and others that a practical database system must enjoy.

Next section introduces DES whereas Section 3 describes its basic implementation of tabling stemmed from [Die87]. Section 4 explains the tabled support for nulls and outer join operations, and Section 5 do the same for duplicates. Finally, Section 6 draws some conclusions and points out some future work.

2 Datalog Educational System

The Datalog Educational System (DES) [SP11] is a free, open-source, multiplatform, portable, in-memory, Prolog-based implementation of a deductive database system. DES 3.0 [SP12] is the next shortcoming release, which enjoys Datalog and SQL query languages, full recursive evaluation with tabling, types, integrity constraints, stratified negation [Ull88], persistency, full-fledged arithmetic, ODBC connections and novel approaches to Datalog and SQL declarative debugging [CGS08, CGS11], test case generation for SQL views [CGS10], null value support, outer join and aggregate predicates and functions [SP11].

DES implements Datalog with stratified negation as described in [Ull88] with safety checks [Ull88, ZCF+97] and source-to-source program transformations for rule simplification, safety and compilation. Evaluation of queries is ensured to be terminating as long as no infinite predicates/operators are considered and since only atomic domains are supported (currently, only the infix operator "`is`" represents an infinite relation).

A reasonable set (for education purposes) of SQL following ISO standard SQL:1999 is sup-

ported (further revisions of the standard cope with issues such as XML, triggers, and cursors, which are outside of the scope of DES). SQL row-returning statements are compiled to and executed as Datalog programs (basics can be found in [Ull88]), and relational metadata for DDL statements are kept. Submitting such a query amounts to 1) parse it, 2) compile to a Datalog program including the relation `answer`/n with as many arguments as expected from the SQL statement, 3) assert this program, and 4) submit the Datalog query `answer`$(X_1,\ldots,X_n)$, where $X_i : i \in \{1,\ldots,n\}$ are n fresh variables. After its execution, this Datalog program is removed. On the contrary, if a data definition statement for a view is submitted, its translated program and metadata do persist. This allows Datalog programs to seamlessly use views created at the SQL side (also tables since predicates are used to implement them). The other way round is also possible if types are declared for predicates.

There are available some usual built-in comparison operators (=, \=, >, ...). When being solved, all these operators demand ground (variable-free) arguments (i.e., no constraints are allowed up to now) but equality, which performs unification. In addition, arithmetic expressions are allowed via the infix operator `is`, which relates a variable/number with an arithmetic expression. The result of evaluating this expression is assigned/compared to the variable. The predicate `not/1` implements stratified negation. Other built-ins include outer joins and aggregates.

3 Tabling-based Query Solving

The computational model of DES follows a top-down-driven bottom-up fixpoint computation with tabling, which follows the ideas found in [SD91, Die87, TS86]. In its current form, it can be seen as an extension of the work in [Die87] in the sense that, in addition, it deals with a modified algorithm for negation, undefined (although incomplete) information, nulls and aggregates. Also, instead of translating each tabled predicate for including fixpoint and memoization management as in [Die87], Datalog rules are stored as dynamic predicates and other predicates explicitly deal with fixpoint computation as shown next.

3.1 Tabling

DES uses an extension table which stores answers to goals previously computed, as well as their calls. For the ease of the introduction, we assume an answer table (`ET` implemented with predicate `et`/1) and a call table (`CT`, implemented with predicate `called`/1) to extensionally store answers and calls, respectively. Also, annotations for completed computations and its handling, which prevents some unnecessary computations, are omitted. Answers may be positive or negative, that is, if a call to a positive goal `G` succeeds, then, the fact `G` is added as an answer to the answer table; if a negated goal `not(G)` succeeds, then the fact `not(G)` is added. Negative facts are deduced when a negative goal is proven by means of negation as failure (closed world assumption (CWA) [Ull88]). Both positive and negative facts cannot occur in a stratifiable program [Ull88]. Calls are also added to the call table whenever they are solved. This allows to detect whether a call has been previously solved and the computed results in the extension table (if any) can be reused. So, repeated answers are not kept in the answer table.

First occurrence during computation of a tabled goal is known as a *generator* whilst further

occurrences of subsumed goals are known as *consumers*. A generator is responsible of building all the *different* answers which will be used by its consumers eventually.

The algorithm implementing this idea (following *ET* algorithm in [Die87]) is depicted next:

```
% Already called. Call table with an entry for the current call
memo(G) :-
  build(G,Q),     % Build in Q the same call with fresh variables
  called(Q),      % Look for a unifiable call in CT for the current call
  subsumes(Q,G), % Test whether CT call subsumes the current call
  !,              %
  et_lookup(G).  % If so, use the results in answer table (ET)
% New call. Call table without an entry for the current call
memo(G) :-
  assertz(called(G)),      % Assert the current call to CT
 ( (et_lookup(G))          % First call returns all previous answers in ET
  ;
   (solve_goal(G),         % Solve the current call using applicable rules
    build(G,Q),            % Build in Q the same call with fresh variables
    no_subsumed_by_et(Q), % Test whether there is no entry in ET for Q
    et_assert(G),          % If so, assert the current result in ET
    et_changed)).          % Flag the change
```

First, test whether there is a previous call that subsumes the current call. For this, `build` constructs a term which is a copy up to variable renaming (i.e., implemented with `copy_term`; more on this later when dealing with nulls). Predicate `subsumes`/2 on the left of Figure 1 implements term subsumption, where a general term subsumes a specific term `st` if grounding of `st` variables (as, e.g., via `numbervars`) makes them unifiable. There are two possibilities: 1) There is such a previous call subsuming the current one: then, use the result in the answer table, if any. To this end, predicate `et_lookup`/1 is implemented as a simple call to the predicate `et`/1 (`et_lookup(G) :- et(G).`) It is possible that there is no such a result (for instance, when computing the goal `p` in the program `p :- p`) and no more tuples can be deduced. 2) Otherwise, process the new call. So, first store the new call in `CT` and then, return all previous answers in `ET` (from a previous fixpoint iteration). Second, solve the goal with the program rules (recursively applying this algorithm). Once the goal has been solved (if succeeded), store the computed answer if there is no any previous answer subsuming the current one (note that, via recursion, we can deliver new answers for the same call). Subsumption is now checked with predicate `no_subsumed_by_et`/1, shown on the right of Figure 1. The whole process is known as a memoization process and will also be referred to as the *memo* function.

3.2 Fixpoint Computation

The memo function is insufficient in itself for computing all possible answers to a goal since incomplete information is used from the goals in its defining rules (as these goals can be mutually

```
subsumes(General,Specific) :-
  \+ \+ (make_ground(Specific),
         General=Specific).
```

```
no_subsumed_by_et(Q,G) :-
  \+ ((et_lookup(Q),
       subsumes(Q,G))).
```

Figure 1: Predicates `subsumes` and `no_subsumed_by_et`

recursive). Therefore, it is needed to ensure that all the possible information is deduced by finding a fixpoint of this function, which is implemented as shown next (following ET^* [Die87]):

```
solve_star(Q,St) :-
  repeat,
  (remove_calls,    % Clear CT
   et_not_changed,  % Flag ET as not changed
   solve(Q,St),     % Solve the call to Q using memoization at stratum St
   fail             % Request all alternatives
  ;
   no_change,       % If no more alternatives, start a new iteration
   !, fail).        % Otherwise, fail and exit
```

First, the call table is emptied to try to obtain new answers for a given call, preserving the previous computed answers. Then, the memo function is applied (via predicate `solve`/2), possibly providing new answers. If the answer table remains the same as before after this last memo function application, we are done. Otherwise, the memo function is reapplied as many times as needed until no changes are found in the answer table. Upon exiting, the answer table contains the meaning of the query (plus perhaps other meanings for the relations used in the computation of the given query).

Predicate `solve` is defined as a straight call to the memo function but for built-ins (that are left apart from the memoization process as would otherwise be a resource waste) and conjunctive goals (which recursively calls itself).

3.3 Dependency Graph and Stratification

Each time a database changes, a predicate dependency graph (PDG) is built [ZCF+97]. This graph shows the dependencies between predicates in the program. Each node in this graph is a program predicate symbol and there are as many nodes as such symbols. Arcs come from each predicate in a rule body (antecedent) to its rule predicate. Arcs are labeled as either negative, if the antecedent node occurs negated, or positive otherwise. This dependency graph is used to looking for a stratification for the program [ZCF+97]. A stratification collects predicates into numbered strata ($1 \ldots N$) so that, given the function $strata(\mathtt{p})$ which assigns a strata number to predicate `p`, then for a positive arc $\mathtt{p} \leftarrow \mathtt{q}$, $strata(\mathtt{p}) \leq strata(\mathtt{q})$, and for a negative arc $\mathtt{p} \overset{\neg}{\leftarrow} \mathtt{q}$, $strata(\mathtt{p}) < strata(\mathtt{q})$. A cycle in this graph containing a negative arc amounts to a non-stratifiable program.

A naïve bottom-up computation would solve all of the predicates in stratum 1, then 2, and so on, until the meaning of the whole program is found. However, the implementation of DES only resort to compute by stratum when a negative dependency occurs in the predicate dependency graph, restricted to the query, as shown next:

```
solve_stratified(Query) :-
  sub_pdg(Query,(_Nodes,Arcs)),
  (neg_dependencies(Arcs) -> solve_star(Query,1)
   ;
    strata(St), sort_by_strata(St,Arcs,Preds),
    build_queries(Preds,Query,Queries), solve_star_list(Queries)).
```

Here, predicate `sub_pdg`/2 gets the current PDG restricted to the query. Predicate `neg_dependencies`/1 tests whether there are negative dependencies in the subgraph. Predicate `stra-`

`ta`/1 gets the current stratification. Predicates in the sub-PDG are sorted w.r.t. this stratification with `sort_by_strata`/2. Then `build_queries` build a list of queries for sorted predicates (an atom with fresh variables for each predicate) appended to the input query. The call to `solve_star_list`/1 solves each of these queries in order by successively calling `solve_star`/2 with each query and its corresponding stratum number.

4 Nulls and Outer Joins

Unknownness has been handled in relational databases long time ago because its ubiquitous presence in real-world applications. Despite its claimed dangers due to unclean semantics (see, e.g., the discussion in [Dat09]), null values to represent unknowns have been widely used. Also, interest in including nulls in logic programming has been stated some time ago [TG94].

Supporting nulls conducts to also provide built-ins to handle them, as outer join operations. DES includes the common outer join operations in relational databases, providing the very same semantics for outer join operators ranging over null values, which are described next.

4.1 Null Semantics

A null value represents unknown data. To include such values into relational database systems (RDBMS's), a new logical value is added for unknown results, leading to a three-valued logic (3VL, for `true`, `false` and `unknown`). Any comparison operator (=, <, ...) relating at least a null value should return the `unknown` logic value [Dat09]. Although a 3VL is assumed for RDBMS's (Oracle, DB2, SQL Server, MySQL, ...), the fact is that the implemented logic does not account for the unknown logic value as it is represented by the null value [Dat09].

However, as we are interested in allowing outer join operations and we rely on a logic engine with 2VL (two-valued logic), we restrict to this, so that any comparison relating at least a null value returns `false` instead of `unknown`. Truth tables for usual logical operators (`not`, `and` and `or`) remain thus as for 2VL. Regarding comparison operators, two (*distinct*) null values are not (known to be) equal, and are (not known to be) distinct. Thus, neither `null = null` (syntactic equality) nor `null \= null` (syntactic disequality) hold. Further, for the same null value, the equality should succeed, as in the conjunctive query `X=null,X=X`. Evaluation of a given expression including at least one null value always returns the same concrete null value. Thus, two expressions are considered equal if they are syntactically equal. This covers, for instance, that the following query succeeds: `X=null,X+1=X+1`.

4.2 Null Representation

Nulls are internally represented with the term `'$NULL'(`*`Id`*`)`, where *`Id`* is a unique integer which does not occur in any other null. This representation is similar to that also suggested in other systems [SWSJ09], but, as a difference, DES considers null as a first class citizen and its internal representation is hidden from the user. Therefore, asserting or consulting a rule as `p(null)` is directly allowed. Since the null value in this rule receives a unique identifier, the conjunctive query `p(X),X=X` succeeds, since `X` stands for the same unknown value (note that

this is in contrast to the flaw in SQL, where `SELECT * FROM p WHERE x=x` discards tuples with a null in `x`).

Any explicit null occurring in either a program or a query is replaced by its internal representation during parsing. Internal representations are also allowed to be written for implementors purposes, but irrespective of the user-provided identifier (which can also be a variable), it is replaced by a unique identifier. Also, when building a new fresh call in predicate `build`/2 (cf. Section 3.1), not only variables have to be fresh, but also any occurrence of a null value. Therefore, this predicate also includes a *null provider* for such occurrences, where concrete null identifiers are replaced by variable identifiers, as `$NULL(V))`, where `V` is a variable. A null provider argument in a rule means that each tuple generated by that rule (and therefore added to the extension table) gets a unique null for that argument eventually. However, along fixpoint iterations, the non-ground null representation is the one to be stored in the extension table. Only once the fixpoint has been reached, nulls are grounded for the answer to be shown to the user. This is in contrast to asserting or consulting a rule containing a null argument, as `p(null)`, where the rule is stored as `p($NULL(N))`, where `N` is a concrete number. Users are precluded from using null generators, which are only available as a result of preprocessing, but it will be needed along the tabled computations of outer joins.

4.3 Outer Join Built-ins

Three outer join operations are provided, following relational database query languages (SQL, extended relational algebra): left (`lj`/3), right (`rj`/3) and full (`fj`/3) outer joins. A left outer join `lj(L,R,C)` computes the cross-product of two relations `L` and `R` that satisfy a third relation `C`, extended with some special tuples including nulls as explained next. Tuples in `L` which have no counterpart in `R` w.r.t. `C` are included in the result, so that the values corresponding to columns of `R` are set to `null`. The right outer join `rj(L,R,C)` is equivalent to `lj(R,L,C)`, and the full outer join `fj(L,R,C)` is equivalent to `lj(L,R,C)` $\cup$ `rj(L,R,C)`. In addition, both `L` and `R` can take the form of such constructions in order to allow more neat, nested applications of outer joins.

In a given cycle of the fixpoint computation for an outer join, a tuple $t_{\mathtt{L}}$ might not find a matching tuple in `R`, but a further cycle may develop new tuples for `R` that do. In order to prevent speculative computations and removing entries from the extension table which are not longer true due to new entries added along fixpoint iterations, the meaning of involved relations in an outer join are required to be computed already before computing the meaning of the outer join. This can be achieved by taking advantage of the stratification idea: relations in outer joins are collected into strata as if they were negative atoms.

4.4 Outer Join Transformations

This section introduces a source-to-source transformation (in a preprocessing phase) for solving the left outer join (other outer joins are analogous), rather than resorting to write (Prolog-)specific code for this. As it is well-known, a single left or right outer join suffices to express others.

A new predicate `$pi` is introduced as an argument of the built-in, void predicate `lj`/1, which does nothing, but is handy to specify a predicate classification in strata. So, the predicate `$pi`

is to be set in a deeper strata than the predicate of the rule in which it occurs, say of predicate p, because the negative arc $p_i \leftarrow p$ is added to the dependency graph. The call `lj($pi)` is solved by predicate `solve`/2 as a built-in, with a straight call to `$pi` (no entries are added to the answer table for `lj`/1). Next, predicate `$pi` is defined to compute the outer join. All of the facts in the meaning of `$pi` come from two sources: the facts in `L` joined with those of `R` that meet `O`, and the facts in `L` joined with nulls that *do not* meet `O`. Next example shows how these data are collected for solving the outer join `v(X,Y) :- lj(s(X,U),t(V,Y),U>V)`:

```
v(X,Y)                                :- lj('$p0'(X,U,V,Y)).
'$p0'(A,B,'$NULL'(C),'$NULL'(D)) :- s(A,B), not('$p1'(A,B,E,F)).
'$p0'(A,B,C,D)                        :- '$p1'(A,B,C,D).
'$p1'(A,B,C,D)                        :- s(A,B), t(C,D), B > C.
```

Predicate `$p0` is source of facts, either provided by the positive case (a *straight* call to `$p1` from the second rule of `$p0`) or by the negative one (a *negated* call to `$p1` in the first rule of `$p0`). This negated call oughts `$p1` to be in a lower strata than `$p0`. Therefore, before computing `$p0`, the meaning of `$p1` is completely available. Predicate `$p1` contains the (possible) hard stuff to be computed since it contains the Cartesian product of two relations, followed by the condition. Despite its arrangement, which may yield to think of a bad computational behavior (compute all tuples from `s`, then all from `t`, and finally filter results), the top-down driven computation looks for a tuple from `s`, then a tuple from `t`, and only adds a new tuple to the answer table of `'$p1'` if the condition `B > C` holds. Indeed, this is quite similar to the RDB implementations of join operations (modulo indexing). The first rule for `'$p0'` builds the null values for the arguments of the right relation `R` for which no tuples are found meeting condition `O`, i.e., So, it is a null provider, as it contains specifications with the form `$NULL(V)`, where `V` is a variable. If there are more than one tuple in `L` that does not match with `R`, each one is therefore joined with a non-ground null tuple. If the null ground representation was instead considered, then the same null tuples will be appended to the result, breaking the assessment that null values should all be unique.

Notice that this transformation includes floundering [BD98] in the first rule for `'$p0'`: the call to `not('$p1'(A,B,E,F))`, where variables `E` and `F` are not range restricted. However, floundering in this concrete case poses no problem as the call to `$p1` is completely computed *before* it is used by any other call and no other negated call occurs in the program. Note that the other call in the program to `$p1` is for the positive case where all of its arguments become ground. In particular, the negated call will use those results and the corresponding negative entries will be added to the answer table. Such negated entries are not be reused by any other (negated) calls in the program because they belong to system-generated predicates. Safety checking takes such floundering into account, avoiding error messages. Other works treat the floundering problem in a more general use of negation (see, e.g., constructive negation [LAC99] and also tabled query evaluation [Dam96]), where non-ground negated calls are possibly involved in recursive calls, which we do not consider in our setting.

Other deductive systems, such as DLV [LPF+06], might benefit from including outer joins as well. In this case, floundering programs are not allowed, but for true negation (CWA is not assumed; instead, negative data are explicitly declared). Fortunately, as pointed out in [Ull88], programs as above can be transformed into non-floundering programs, where all calls to negated

goals are ensured to be ground. Next program shows this transformation, where non-relevant variables are dropped and unfolding is applied:

```
v(X,Y)                              :- '$p0'(X,U,V,Y).
'$p0'(A,B,'$NULL'(C),'$NULL'(D)) :- s(A,B), not('$p1'(B)).
'$p0'(A,B,C,D)                      :- s(A,B), t(C,D), B > C.
'$p1'(B)                            :- s(A,B), t(C,D), B > C.
```

However, comparing this version to the example, even when the number of relations does not increase, extra computation has to be done in the second clause of `$p0`. So, although it seems possible to compute outer joins in DLV with this technique, nulls should be natively supported; otherwise it couldn't be applied because there is no provision to get unique identifiers for null values in this system (DLV does not feature a general-purpose programming language, but a deductive language).

XSB [SSW94b] is another system which supports non-ground semantics allowing floundering programs with the use of the special negation `sk_not`/1, which automatically produces a similar translation as explained before [SWSJ09]. To write outer joins in this system, in particular it is needed to generate unique identifier integer numbers for the null values and declare as tabled the predicates involved in the computation of the outer join. The following program implements the outer join example in XSB:

```
:- table('$p0'/4), table('$p1'/4), table(s/2), table(t/2).
main(Vs) :- findall(v(X,Y),v(X,Y),Vs).
v(X,Y)   :- '$p0'(X,U,V,Y).
'$p0'(A,B,'$NULL'(C),'$NULL'(D)) :-
    get_id(C), get_id(D), s(A,B), sk_not('$p1'(A,B,E,F)).
'$p0'(A,B,C,D) :- '$p1'(A,B,C,D).
'$p1'(A,B,C,D) :- s(A,B), t(C,D), B > C.
:- dynamic id/1.
id(0).
get_id(X) :- id(X), retractall(id(X)), Y is X+1, assertz(id(Y)).
```

Here, the main entry point (predicate `main`/1) returns a list of deduced facts via the metapredicate `findall`, which collects all answers to the goal `v(X,Y)`. Predicate `get_id` returns a new integer each time it is called, therefore allowing to uniquely identify nulls.

5 Duplicates

Allowing tables to contain duplicate rows and queries returning also duplicates is a common feature in current RDBMS's. However, the introduction of duplicates is claimed to suffer some other issues: Duplicates in a table are repeated rows in a relation and, from a logical viewpoint, have no sense because repeated rows mean the same[1]. Another issue with duplicates is that equivalent-intended statements can deliver a different number of duplicate rows [Dat09]. As well, they preclude query optimizations and make optimizers much more complicated than if were if no duplicates were allowed. Nonetheless, duplicates are useful in a number of situations, for instance, when considering aggregates.

Noticeably, whilst in relational databases they are assumed, they are not usual in deductive databases (mainly because of the claimed issues), where they are removed by default. However,

[1] Citing Codd: "If something is true, saying it twice doesn't make it any more true."

there are deductive systems supporting duplicates as LDL++, but duplicates are removed from recursive rules. As a main difference, DES also allows recursive rules to be generators of duplicates in a similar way as in SQL recursive statements. Since duplicates are not removed from derived relations, each rule is understood as a possible, distinct duplicate generator. When duplicates are disabled, they are discarded along computation, i.e., subsumed answers are not added to the answer table. Next subsection shows how this behavior is supported in a tabled system.

5.1 Duplicates and Tabling

An alternative for supporting duplicates in extensional predicates it to distinguish each rule in the program, so that two repeated rules are not considered to be equivalent w.r.t. subsumption. To this end, we can add a unique *rule identifier* to each rule in the program, as in the program `1:p(a), 2:p(a), 3:p(b).` Then, an entry in the answer table will contain tuples of the form (*Atom*, *RuleId*), where *Atom* is the answer and *RuleId* the rule identifier that generated that answer. In this example, the entries obtained for the call `p(X)` are: {(`p(a)`,`1`), (`p(a)`,`2`), (`p(b)`,`3`)}. So, the first and second answers do not subsume each other and the usual behavior for answer subsumption can be kept. From a user viewpoint, rule identification is of no use (as in relational databases), so that they are hidden from the user when displaying answers and listing rules.

As an example of a recursive predicate, let's consider the rules {`p(a)`, `p(a)`, `p(X):-p(X)`} (two facts and a recursive rule, respectively). Its intended meaning is the multiset containing the tuple `p(a)` four times, where two tuples correspond to the extensional rules for `p` and the other two to the single intensional rule for `p`. This intensional rule generates one tuple for each extensional rule. By adding the rule `p(X) :- p(X)` once more, the meaning of `p` would contain `p(a)` ten times, i.e., it contains: two tuples from the two facts, and four tuples for each recursive rule. The first recursive rule is source of four tuples because of the two facts and the two tuples from the second recursive rule (analogously for the second recursive rule). In fact, this mimics SLD resolution by collecting all possible answers coming from *different* sources, therefore pruning infinite computation paths, i.e., all possible computation paths are considered, stopping when a (recursive) node already used in the computation is reached. Figure 2 shows the tabling tree for the query `p(X)` and the annotated program `1.p(a)`, `2.p(a)`, `3.p(X) :- p(X)`, and `4.p(X) :- p(X)`. Infinite computations are elided in this case because the rule with identifier `3` is neither reused for solving its body nor the body of rule `4` as this rule is a descendant of `3` (analogously for rule `4`).

The idea above about uniquely identifying each rule can be applied to recursive predicates as well. One possible solution is to keep track of the computation path by annotating the rules which have been used to deduce a given atom. So, each entry in the extension table can be identified by a chain of such rule identifiers. Each deduced atom is associated with a pair (*Id*, *IdChain*), where *Id* is the identifier of the rule which is the generator of the atom, and *IdChain* is the list of as many pairs as goals in rule *Id*. So, all rule identifiers in the computation path up to the tabling tree leaf are stored. Referring to Figure 2, the following are the entries stored in the answer table after computation, from left to right: (`p(a)`, (`1`,[])), (`p(a)`, (`2`,[])), (`p(a)`, (`3`,[(`1`,[])])), ..., (`p(a)`, (`4`,[(`3`,[(`2`,[])])])).

To implement this, two additional parameters to predicate `memo`/2 are added: `D` for selecting

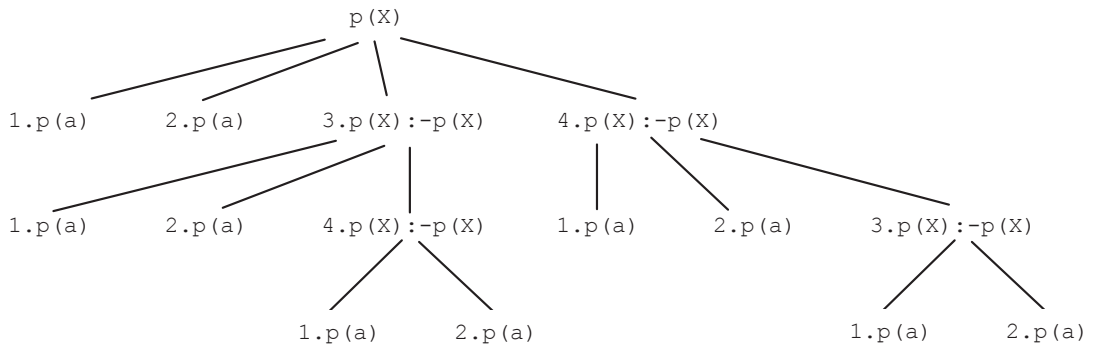


Figure 2: Tabling tree for the query `p(X)`

whether or not duplicate answers for a relation are requested (`all`, `distinct`, resp.) or eliminate duplicates for a given set of arguments (`distinct(Vs,Ps)`), and `Id` as the identifier pair as described above. Its new prototype is `memo(+G,+St,+D,-Id)`. These two parameters are passed to `et_lookup` (which is further explained in Subsection 5.3) in the first clause of this predicate and also to `no_subsumed_by_et(Q,D,(G,Id))`, which is modified as follows:

```
% no_subsumed_by_et(+Query,+Distinct,+(Goal,IdGoal))
no_subsumed_by_et(Q,all,(_G,_IdG)) :- % No entry matching Q
  \+ et_lookup(Q,all,_IdQ), !.
no_subsumed_by_et(Q,all,(G,IdG)) :-   % Existing entries matching Q
  nr_id(IdG),  % Check that IdG is not cyclic
  \+ (et_lookup(Q,all,IdQ), my_subsumes((Q,IdQ),(G,IdG))).
no_subsumed_by_et(Q,_D,(G,_IdG)) :-   % For distinct answers
  \+ (et_lookup(Q,all,_IdQ), my_subsumes(Q,G)).
```

5.2 Duplicates and Nulls

When duplicates are disabled, a null value is considered as a single constant in answers, so that repeated entries are removed. For example, if an answer contains two occurrences of `p(null)`, only one is shown. This follows the SQL criterium when applying the clause `DISTINCT` in a `SELECT` statement. If duplicates are enabled, each null is considered as a different constant identified by its internal numeric representation. Thus, in this example, the user would obtain two tuples. However, tabling computes both tuples in both cases. By enabling development listings, which in particular shows null internal representations, in the former case both tuples are listed.

5.3 Duplicate Elimination

When duplicates are enabled, duplicate elimination is provided with the built-in `distinct/1`, which applies to a positive atom. In this case, duplicate elimination policy for an atom `A` consists of discarding all duplicates for `A` and all of its descendants in the tabling tree. To implement this policy, function `memo` is added with a parameter indicating whether duplicates are discarded

(distinct) or not (all). When a distinct(*G*) call is to be solved with predicate solve, a call to this function is called with this parameter, which is passed to subsequent descendant calls to the memo function in the same subtree. Because there can be other calls to A not involved in a duplicate elimination path, all its answers are computed by the generator, and consumers involved in duplicate elimination are responsible of using non-repeated entries. So, each consumer uses the modified predicate et_lookup, as shown next, including a new second parameter selecting whether or not all duplicate answers are requested:

```
% et_lookup(+Goal,+Distinct,-Id)
et_lookup(G,all,IdG)      :- et(G,IdG).
et_lookup(G,distinct,IdG) :-
  findall((G,IdG),et(G,IdG),GIdGs), setof(G,IdG^member((G,IdG),GIdGs),Gs),
  member(G,Gs), once(member((G,IdG),GIdGs)).
```

This predicate is called from predicate memo/4 in two scenarios: First, for consumers that get all entries in the answer table for a subsumed call (first clause of memo) or from a previous fixpoint iteration (first call in second clause of memo), both with second argument of et_lookup with distinct. In this first case and when distinct answers are required, second clause of et_lookup is applied. Second case is for generators (second call in second clause of memo), where all answers are required and this predicate is called with second argument with all. For providing distinct entries in ET, first all entries matching the input goal are collected with findall, along with their identifiers. Next, setof filters duplicated goals. Each possible answer G is selected with member via backtracking. One class representant is finally selected with the only one solution call to member in last line. Its identifier of this representant is the one returned.

5.4 Duplicates and Projection

Aforementioned handling of duplicate elimination is not enough to deal with correlated SQL queries. Compiling such a SQL query to Datalog may involve a distinct operator for a projection (a subset) of the arguments in a relation as, for instance:

```
CREATE TABLE t(a int, b int);  CREATE TABLE s(a int, b int);
       CREATE VIEW v(a) AS SELECT a FROM t WHERE a IN
         (SELECT DISTINCT a FROM s WHERE t.b<s.b);
```

A possible Datalog program for this view follows:

```
v(A)     :- t(A,B), distinct(v_1(A,B)).
v_1(A,B) :- s(A,C), B < C.
```

But this is not a correct equivalent formulation because distinct/1 applies to different tuples from v_1 instead of different values for a, and b must be passed to v_1 to filter results. So, a new built-in distinct(*Arguments*,*Relation*) is needed, which computes different results for the list of relation arguments for the relation. The first Datalog rule would be rewritten as v(A) :- t(A,B), distinct([A],v_1(A,B)).

Solving this new built-in is via the call memo(G,St,R,distinct(Vs,Ps),Id), where G is to be computed for returning only distinct tuples of variables Vs, which correspond to

argument positions `Ps`. These positions are kept to account projection positions as any variable in `Vs` might become ground before solving this call. `no_subsumed_by_et`/3 is modified accordingly by adding the next clause, which deals with duplicate elimination when duplicates are enabled.

```
no_subsumed_by_et(Q,distinct(_Vs,Ps),(G,_IdG)) :-
  \+ (functor(Q,F,A), functor(FQ,F,A),
      get_ith_arg_list(Ps,Q,QAs), get_ith_arg_list(Ps,FQ,QAs),
      et(FQ,_Id),get_ith_arg_list(Ps,G,GAs),my_subsumes(QAs,GAs)).
```

As well, predicate `et_lookup`/3 is added with the following clause:

```
et_lookup(G,distinct(Vs,_SG),IdG) :-
  findall((G,IdG),et(G,IdG),GIdGs), term_variables(G,GVs),
  set_diff(GVs,Vs,EVs),
  build_ex_quantifier(EVs,my_member((G,IdG),GIdGs),QG),
  setof(Vs,IdG^QG,Vss), my_member(Vs,Vss), once(my_member((G,IdG),GIdGs)).
```

Although similar to the second clause of this predicate in previous section, this clause adds handling of unprojected variables by building existential quantifiers over them, so that `setof`/3 returns distinct tuples for projected variables.

5.5 Duplicates in Aggregates

Aggregates are supported in DES in several flavors [SP11], both as functions and predicates, but for the sake of this paper, we restrict the presentation to a simplified aggregate predicates. An aggregate predicate returns its result in its last argument position, as in `sum(p(X),X,R)`, which binds `R` to the cumulative sum of values for `X`, provided by relation `p`. Duplicate elimination versions are also available, such as `sum_distinct`/3.

Solving rules involving aggregates via successive fixpoint iterations might lead to incorrect entries to be added to the extension table because in a given iteration it is not ensured that all the meaning of the aggregated relation is computed. A straightforward approach to solve them is analogous to negation: new negative arcs are added to the dependency graph in order to place such relation in a stratum lower than the predicate of the rule in which it occurs. So, the rule `s(R) :- sum(p(X),X,R)` implies an arc $\mathtt{s} \overset{\neg}{\leftarrow} \mathtt{p}$, which in turn also implies the constraint $\mathit{strata}(\mathtt{p}) < \mathit{strata}(\mathtt{s})$ for the stratification.

Then, when solving an aggregate in a given fixpoint iteration, all the tuples of its aggregated relation are known, so that duplicate elimination is simply performed by applying `setof` instead of `bagof` to groups from entries in the answer table. A simplified version implementing this is shown next:

```
compute_distinct_aggregate_pred(Aggr) :-
  Aggr =.. [F,R,V,O], R =.. [_P|Args], get_arg_position(V,Args,I),
  nf_setof(N, CR^Ids^(et(CR,Ids),\+ \+ (CR=R),
                      arg(I,CR,N),N\='$NULL'(_Id)), Ns),
  compute_aggregate(F,Ns,O).
```

Here, `nf_setof` is the non-failing version of `setof` (it returns an empty list instead of failure), and collects all values for the argument to which the aggregate is applied (`X` in the example above). As in relational databases, null arguments are omitted from this set.

6 Conclusions

This paper has shown how to include null, outer joins and duplicates altogether into a tabled deductive database system. Since this system compiles SQL statements to Datalog programs, it was a need to embody such features coming from relational database systems into the concrete deductive inference engine DES. Because this system is not geared towards performance, this implementation should be seen as a proof of concept. Some hints have been provided for the porting to other deductive systems and future work may include to use other external efficient engines such as XSB.

Bibliography

[BD98] G. Brewka, J. Dix. Knowledge Representation with Logic Programming. In Dix et al. (eds.), *Proceedings of LPKR'97*. LNAI 1471, pp. 1–51. Springer-Verlag, 1998.

[CGS08] R. Caballero, Y. García-Ruiz, F. Sáenz-Pérez. A Theoretical Framework for the Declarative Debugging of Datalog Programs. In *International Workshop on Semantics in Data and Knowledge Bases*. LNCS 4925, pp. 143–159. Springer, 2008.

[CGS10] R. Caballero, Y. García-Ruiz, F. Sáenz-Pérez. Applying Constraint Logic Programming to SQL Test Case Generation. In *Proc. International Symposium on Functional and Logic Programming (FLOPS'10)*. LNCS 6009. 2010.

[CGS11] R. Caballero, Y. García-Ruiz, F. Sáenz-Pérez. Algorithmic Debugging of SQL Views. In *Ershov Informatics Conference (PSI'11)*. LNCS. Springer, 2011. In Press.

[Cod70] E. Codd. A Relational Model for Large Shared Databanks. *Communications of the ACM* 13(6):377–390, June 1970.

[Dam96] C. Damásio. *Paraconsistent Extended Logic Programming with Constraints,*. PhD thesis, Dept. de Informâtica, Universidade Nova de Lisboa, 1996.

[Dat09] C. J. Date. *SQL and relational theory: how to write accurate SQL code*. O'Reilly, Sebastopol, CA, 2009.

[Die87] S. W. Dietrich. Extension Tables: Memo Relations in Logic Programming. In *IEEE Symp. on Logic Programming*. Pp. 264–272. 1987.

[GCH+08] P. C. de Guzmán, M. Carro, M. V. Hermenegildo, C. Silva, R. Rocha. An improved continuation call-based implementation of tabling. PADL'08, pp. 197–213. Springer-Verlag, Berlin, Heidelberg, 2008.

[LAC99] J. Y. Liu, L. Adams, W. Chen. Constructive negation under the well-founded semantics. *JLP* 38(3):295–330, 1999.

[LPF+06] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM Tran. on Computational Logic* 7(3):499–562, 2006.

[RSC05] R. Rocha, F. M. A. Silva, V. S. Costa. Dynamic Mixed-Strategy Evaluation of Tabled Logic Programs. In Gabbrielli and Gupta (eds.), *ICLP*. LNCS 3668, pp. 250–264. Springer, 2005.

[RU93] R. Ramakrishnan, J. Ullman. A survey of research on Deductive Databases. *JLP* 23(2):125–149, 1993.

[SP11] F. Sáenz-Pérez. DES: A Deductive Database System. *Electronic Notes on Theoretical Computer Science* 271:63–78, March 2011.

[SP12] F. Sáenz-Pérez. Datalog Educational System. May 2012. http://des.sourceforge.net/.

[SD91] C. Shih, S. Dietrich. Extension Table Evaluation of Datalog Programs with Negation. In *Proc. of the IEEE International Phoenix Conference on Computers and Communications*. Volume AZ, pp. 792–798. Scottsdale, March 1991.

[SS06] Z. Somogyi, K. Sagonas. Tabling in mercury: Design and implementation. In *In Proceedings of Practical Aspects of Declarative Programming (PADL'06)*. Pp. 150–167. Springer-Verlag, 2006.

[SSW94a] K. Sagonas, T. Swift, D. S. Warren. XSB as an Efficient Deductive Database Engine. In *In Proceedings of the ACM SIGMOD International Conference on the Management of Data*. Pp. 442–453. ACM Press, 1994.

[SSW94b] K. Sagonas, T. Swift, D. S. Warren. XSB as an efficient deductive database engine. In *SIGMOD'94: Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*. Pp. 442–453. ACM, New York, NY, USA, 1994.

[SW10] T. Swift, D. S. Warren. XSB: Extending Prolog with Tabled Logic Programming. *CoRR* abs/1012.5123, 2010. Submitted to TPLP.

[SWSJ09] T. Swift, D. Warren, K. Sagonas, J. Freire et al. The XSB System Version 3.2. Volume 2: Libraries, Interfaces and Packages. 2009. `http://xsb.sourceforge.net/`.

[TG94] B. Traylor, M. Gelfond. Representing Null Values in Logic Programming. In *LFCS'94*. Pp. 341–352. 1994.

[TS86] H. Tamaki, T. Sato. OLDT Resolution with Tabulation. In *Third International Conference on Logic Programming*. Pp. 84–98. 1986.

[Ull88] J. D. Ullman. *Database and Knowledge-Base Systems, Vols. I (Classical Database Systems) and II (The New Technologies)*. Computer Science Press, 1988.

[ZCF+97] C. Zaniolo, S. Ceri, C. Faloutsos, R. T. Snodgrass, V. S. Subrahmanian, R. Zicari. *Advanced Database Systems*. Morgan Kaufmann, 1997.

[ZS03] N.-F. Zhou, T. Sato. Efficient fixpoint computation in linear tabling. PPDP'03, pp. 275–283. ACM, New York, NY, USA, 2003.

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): Actas de las "XII Jornadas sobre Programación y Lenguajes (PROLE'2012), Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

Test-Case Generation for SQL Nested Queries with Existential Conditions

Rafael Caballero[1] * **José Luzón-Martín** [2] **Antonio Tenorio** [3]

[1] rafa@sip.ucm.es [2] jose.11.10.88@gmail.com [3] senrof21@gmail.com
Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain

Abstract: This paper presents a test-case generator for SQL queries. Starting with a set of related SQL views that can include existential subqueries in the conditions, the technique finds a database instance that can be used as a test-case for the target view. The proposal reduces the problem of generating the test-cases to a Constraint Satisfaction Problem using finite domain constraints. In particular, we present a new approach for existential conditions that makes possible to find test-cases for a wider set of queries. The soundness and correctness of the technique with respect to a simple operational semantics for SQL queries without aggregates is proven. The theoretical ideas have been implemented in an available prototype.

Keywords: SQL, Test-cases, constraints, finite domains

1 Introduction

This paper presents a technique for producing test-cases that allows the checking of a set of correlated SQL [SQL92] views. Test-cases are particularly useful in the context of databases, because checking if queries which are defined over real-size database instances are valid is a very difficult task. Therefore, the goal is to automatically obtain a small database instance that allows the user to readily check if the defined SQL views work as expected. There are many different possible coverage criteria for test-cases (see [ZHM97, AO08] for a general discussion). In the particular case of SQL [CT04, ST09], it has been shown that good criteria like the Full Predicate Coverage (FPC) can be obtained by transforming the initial SQL query into a set of independent queries, where each one is devoted to checking a particular component of the query and then obtaining a *positive test-case* for each query in this set. A positive test-case is a test case following the simple criterium of being non-empty database instances such that the query/relation studied produces a non-empty result. As proposed in [CGS10], negative test-cases can be defined in terms of positive test-cases, thus, negative test cases can also be obtained with a positive test-case generator. A first step towards a solution was presented in [CGS10], where the problem was reduced to a Constraint Satisfaction Problem (CSP in short). Although self-contained, this paper must be seen as an improvement with respect to this previous work. More specifically, the main contribution is that queries including existential subqueries are allowed. These subqueries are introduced in SQL conditions by means of the reserved word exists and play an important role in

* Work partially supported by the Spanish projects STAMP (TIN2008-06622-C03-01), Prometidos-CM (S2009TIC-1465) and GPD (UCM-BSCH-GR35/10-A-910502)

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): PROLE'2012, pp. 103-117, ISBN:978-84-15487-27-2.
Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

defining SQL views. In [CGS10] the treatment of these subqueries was very limited (only if the exists occurred as part of a conjunction at the outer level). Moreover, the operational semantics used in that work (the Extended Relation Algebra, ERA in short) does not allow subqueries, a very important feature that could not be included in the theoretical results. We overcome this limitation in the current paper by:

1. Defining a new SQL Operational Semantics (called SOS), presented in Section 2 that allows existential subqueries as part of the conditions in the where clause. We prove the equivalence of SOS and ERA over the set of basic queries without aggregates.
2. Defining a CSP that includes representation of existential subqueries (Section 3), and proving its correctness and completeness.
3. Implementing a new prototype (Section 4) that generates the test-cases using a constraint solver.

The work is concluded in Section 5, which summarizes the results and discusses future work.

2 SQL operational semantics

A *table schema* is of the form $T(A_1,\ldots,A_n)$, with T the table name and A_i attribute names for $i = 1\ldots n$. We will refer to a particular attribute A by using the notation $T.A$. Each attribute A has an associated type (*integer*, *string*, ...) represented by $type(T.A)$. In this paper we only consider *integer* types, although the inclusion of other data types will be considered in future work.

An *instance* of a table schema $T(A_1,\ldots,A_n)$ will be represented as a finite multiset of functions (called rows) $\{|\mu_1,\mu_2,...,\mu_m|\}$ such that $dom(\mu_i) = \{T.A_1,\ldots,T.A_n\}$, and $\mu_i(T.A_j) \in type(T.A_j)$ for every $i = 1,\ldots,m$, $j = 1,\ldots,n$. Observe that we qualify the attribute names in the domain by table names. This is done because in general we will be interested in rows that combine attributes from different tables, usually as result of cartesian products. In the following, it is useful to consider each attribute $T.A_i$ in $dom(\mu)$ as a logic variable, and μ as a logic substitution.

The concatenation of two rows μ_1, μ_2 with disjoint domain is defined as the union of both functions represented as $\mu_1 \odot \mu_2$. Given a row μ and an expression e we use the notation $e\mu$ to represent the value obtained applying the substitution μ to e.

If $dom(\mu) = \{T.A_1,\ldots,T.A_n\}$ and $\nu = \{U.A_1 \mapsto T.A_1,\ldots,U.A_n \mapsto T.A_n\}$ (i.e., ν is a table renaming) we will use the notation μ^U to represent the substitution composition $\nu \circ \mu$. The previous concepts for concatenations and substitutions can be extended to multisets of rows in a natural way. For instance, given the multiset of rows S and the row μ, $S\mu$ represents the application of μ to each member of the multiset. Analogously, given two multisets S_1, S_2, we define

$$S_1 \odot S_2 = \{\nu \circ \mu \mid \nu \in S_1, \mu \in S_2\}$$

which constitutes the natural representation of *cartesian products* in relational databases. Observe that although our representation for rows is different from the representation used usually in relational databases (tuples), it is possible to define an isomorphism $\ll \cdot \gg$ between the two representations. If R is a relation and $\mu \in R$ is defined as $\mu = \{|T_1.A_1 \mapsto a_1,\ldots T_n.A_n \mapsto a_n|\}$,

then $\ll\mu\gg = (a_1,\ldots,a_n)$ where the order in the tuple is the lexicographic order of the attribute names. In order to define the inverse transformation we assume that every relation has a well-defined schema with attribute names $(T_1.A_1,\ldots,T_n.A_n)$ with the attribute name positions in lexicographic order. Obviously, if R is either a table or a view, $T_i = R$ for $i = 1\ldots n$, but in the case of queries we can assume different relation names as prefixes. Then if $t = (a_1,\ldots,a_n)$ is a tuple in R we define $\ll\mu\gg^{-1} = T_1.A_1 \mapsto a_1,\ldots T_n.A_n \mapsto a_n$. It can be checked that the transformation is distributive with respect to $\prod$ and $\odot$, this means that $\ll t_1 \times t_2\gg^{-1} = \ll t_1\gg^{-1} \odot \ll t_2\gg^{-1}$ and $\ll\mu_1 \odot \mu_2\gg = \ll\mu_1\gg \times \ll\mu_2\gg$.

Definition 1 SQL view and query syntax.
The general form of a view is: create view V($A_1,\ldots,A_n$) as Q, with Q a query and $V.A_1,\ldots V.A_n$ the name of the view attributes. Queries can be:

1. Basic queries Q = select $e_1,\ldots,e_n$ from $R_1\ B_1\ ,\ldots,R_m\ B_m$ where C;

   with R_j tables or views for $j = 1\ldots m$, e_i, $i = 1\ldots n$ expressions involving constants, predefined functions and attributes of the form $B_j.A$, $1 \le j \le m$, and A an attribute of R_j. The condition C must have one of the following forms:

   $$C ::= \mathsf{false} \mid \mathsf{true} \mid e_1 \diamond e_2 \mid C_1 \text{ and } C_2 \mid C_1 \text{ or } C_2 \mid \text{not} C1 \mid \mathsf{exists}(\mathsf{Q})$$

   with e_1, e_2 arithmetic expressions, $\diamond \in \{=, <>, >, <, >=, <=\}$, C_1, C_2 SQL conditions and Q an SQL query.

   In the rest of the paper the logic values *true* and *false* are represented respectively by $\top$ and $\bot$ in order to be distinguished from the SQL reserved words true and false.

2. Unions of the form Q_1 union Q_2, with Q_1 and Q_2 queries.

3. Intersections of the form Q_1 intersects Q_2, with Q_1 and Q_2 queries.

A *database schema* D is a tuple $(\mathscr{T},\mathscr{C},\mathscr{V})$, where $\mathscr{T}$ is a finite set of tables, $\mathscr{C}$ a finite set of database constraints and $\mathscr{V}$ a finite set of views. A *database instance* d of a database schema is a set of table instances, one for each table in $\mathscr{T}$ verifying $\mathscr{C}$ (thus we only consider *valid instances*). To represent the instance of a table T in a database instance d we will use the notation $d(T)$. In this paper *primary key* and *foreign key* constraints are considered. They can be formally defined as follows:

Definition 2 Let D be a database schema and d a database instance. Let T be a table in D such that $d(T) = \{|\mu_1,\ldots,\mu_n|\}$, then:

1. If T has a primary key constraint defined over the columns $C_1,\ldots,C_p$, then we say that d is a valid instance with respect to this constraint if for every $i = 1\ldots n$ the logic formula

   $$\bigwedge_{j=1, j\neq i}^{n} \left(\bigvee_{k=1}^{p} \mu_i(T.C_k) \neq \mu_j(T.C_k)\right)$$

   holds (that is, it is evaluated to $\top$).

2. If T has a foreign key constraint for the columns $C_1,\ldots,C_f$ referring the columns $C'_1,...C'_f$ from table T_2. Then suppose that $d(T_2) = \{|v_1,\ldots,v_{n'}|\}$. Then d is a valid instance with respect to this foreign key if for every μ_i, $i = 1\ldots n$ the following logic formula holds:

$$\bigvee_{j=1,j\neq i}^{n'} (\bigwedge_{k=1}^{f} \mu_i(T.C_k) = v_j(T_2.C'_k))$$

That is, in the case of the primary key we require that every pair of rows must differ in at least one attribute of the primary key. In the case of the foreign key, every row μ_i in T requires the existence of another tuple v_j in T_2 such that they have the same values over the attributes defining the primary case in each case.

A *symbolic database instance* d_s is a database instance whose rows can contain logic variables. We say that d_s is satisfied by a substitution μ when $(d_s\mu)$ is a database instance. μ must substitute all the logic variables in d_s by domain values.

Next we present a small example that is used in the rest of the paper.

Example 1 *A board game for multiple players. Players are included into the following table*

```
create table player {  id int primary key   };
```

The board can contain pieces from different players, but only one piece at each position. The board state is represented by the following table:

```
create table board {
  int x,
  int y,
  int id,
  primary key x,y;
  foreign key(id) references player(id);
  };
```

where x, y *are the position in the board and* id *the player identifier.*

We are interested in detecting the ids *of the players that are still playing (that is, they have at least one piece on the board), and all their pieces are* threatened*: for every piece of the player there is a piece of a different player which is either in the same* x *or in the same* y*. The views defining such pieces are the following:*

```
create view nowPlaying(id) as
  select p.id
  from player p
  where exists (select b.id from board b where b.id=p.id);

create view checked(id) as
  select p.id
  from player p
  where exists (select n.id from nowPlaying n where n.id = p.id) and
        not exists (select b1.id from board b1
```

```
where b1.id = p.id and
      not exists
     (select b2.id from board b2
      where (b2.x - b1.x) * (b2.y-b1.y)=0 and
            (b1.id <> b2.id)));
```

The *syntactic dependency tree* for a view, table or query R, with R in the root of this tree, is defined as:

- If R is a table it has no children.
- If R is a view the only child is the syntactic dependency tree of its associated query.
- If R is a query the children are the syntactic dependency trees of the relations occurring in the from section, plus one child for each subquery occurring in the where section.

Now we are ready to define our SQL operational semantics (SOS for short).

Definition 3 Let $\mathscr{D}$ a relational database schema and d be an instance of $\mathscr{D}$. Then the SQL operational semantics, $SOS(d)$ is defined as follows:

1. For any table T defined in $\mathscr{D}$, $\langle T,d\rangle = d(T)$.
2. For any simple query Q

   select $e_1, \ldots, e_n$ from $R_1\ B_1\ , \ldots, R_m\ B_m$ where C; define:

   $$\langle Q,d\rangle = \{s_Q(\mu) \mid \nu_1 \in \langle R_1,d\rangle, \ldots, \nu_m \in \langle R_m,d\rangle, \mu = {\nu_1}^{B_1} \odot \cdots \odot {\nu_m}^{B_m}, \langle C\mu,d\rangle\}$$

   where $s_Q(\mu) = \{Q.A_1 \mapsto (e_1\mu), \ldots, Q.A_n \mapsto (e_n\mu)\}$, and $\langle C,d\rangle$ is defined as follows

   - If $C \equiv$ false then $\langle C,d\rangle = \bot$
   - If $C \equiv$ true then $\langle C,d\rangle = \top$
   - If $C \equiv e$, with e an arithmetic expression involving constants then $\langle C,d\rangle = e$
   - If $C \equiv e_1 \diamond e_2$, with $\diamond$ a relational operator, then $\langle C,d\rangle = (\langle e_1,d\rangle \diamond \langle e_2,d\rangle)$.
   - If $C \equiv C_1$ and C_2 then $\langle C,d\rangle = \langle C_1,d\rangle \wedge \langle C_2,d\rangle$
   - If $C \equiv C_1$ or C_2 then $\langle C,d\rangle = \langle C_1,d\rangle \vee \langle C_2,d\rangle$
   - If $C \equiv$ not C_1 then $\langle C,d\rangle = \neg\langle C,d\rangle$
   - If $C \equiv$ exists Q then $\langle C,d\rangle = \langle Q,d\rangle \neq \emptyset$
3. If Q is a view with attributes $A_1,\ldots,A_n$ defined by a query of the form Q_1 union Q_2, then $\langle Q,d\rangle = \langle Q1,d\rangle \cup \langle Q2,d\rangle$ with $\cup$ the union multiset operator.
4. Analogously, if Q is of the form Q_1 intersection Q_2 then, $\langle \mathscr{R},d\rangle = \langle Q1,d\rangle \cap \langle Q2,d\rangle)$, with $\cap$ the intersection multiset operator.
5. For any view V with definition create view V($E_1, \ldots, E_n$) as Q, $\langle V,d\rangle = \langle Q,d\rangle\{V.E_1 \mapsto Q.A_1,\ldots,V.E_n \mapsto Q.A_n\}$.

The following observations can be useful for understanding SOS. Queries are treated as anonymous views, and the name Q is always given as table name to the rows in the result. Analogously A_i is always the name given to the i-th expression in the select clause. This is useful for instance for ensuring that unions and intersections provide homogeneous rows, and also for the renaming applied by views in the last item. The expression $(e_1 \diamond e_2)\mu$ in the second case represents the logic value $\top$ or $\bot$ obtained after replacing each attribute by its value in μ and evaluating the relational expression in usual logic. In well-defined SQL queries all the attributes in the expressions correspond exactly to one attribute of a relation defined in the query. Well-defined existential subqueries must verify that after replacing the external attributes by values they become well-defined queries. This property is exploited in the definition used in the existential condition $C(\mu) = (\langle Q\mu, d\rangle \neq \emptyset)$, which must be read as: first replace in Q all the external attributes by their values, then obtain their semantics, and finally check whether the result is different from the empty multiset.

Using this definition we can define a *positive test-case* (PTC) for a view V as a non-empty database instance d such that $\langle V\rangle \neq \emptyset$. In previous papers we have used the Extended Relational Algebra (ERA from now on) [GUW08] as suitable operational semantics for SQL. However ERA does not allow existential subqueries, a key point of this work that is solved in SOS. Defining ERA and the transformation of SQL into ERA is beyond the scope of this paper, and we only mention some of the basic operations:

- Unions and intersections. The union of R and S, is a multiset $R \cup S$ in which the row μ occurs $n+m$ times. The intersection of R and S, $R \cap S$, is a multiset in which the row μ occurs $min(n,m)$ times.

- Projection. The expression $\pi_{e_1 \mapsto A_1,\ldots,e_n \mapsto A_n}(R)$ produces a new relation producing for each row $\mu \in R$ a new row $\{A_1 \mapsto e_1 \ll \mu \gg, \ldots, A_n \mapsto e_n \ll \mu \gg\}$. That is, we substitute in each e_i the attribute names by their values in μ and then evaluate e_i. The resulting multiset has the same number of rows as R.

- Selection. Denoted by $\sigma_C(R)$, where C is the condition that must be satisfied for all rows in the result. The selection operator on multisets applies the selection condition to each row occurring in the multiset independently.

- Cartesian products. Denoted as $R \times S$, each row in the first relation is paired with each row in the second relation.

- Renaming. The expression $\rho_S(R)$ changes the name of the relation R to S.

A sound requirement is that both ERA and SOS must define the same semantics over the common SQL fragment. This is important because ERA is assumed as the standard SQL semantics.

Theorem 1 *Let D be a database schema and d a database instance. Let R be a relation in D or a query defined over relations in D. Assume that the relations in the syntactic dependency tree of R do include neither aggregates not existential subqueries. Then* $\eta \in \langle R, d\rangle$ *with cardinality k iff* $\ll \eta \gg \in ERA(R,d)$ *with cardinality k.*

Proof. Using induction on the depth of the syntactic dependency tree for R, and distinguishing cases depending on the form of R:
- R is a query of the form select $e_1, \ldots, e_n$ from $R_1\ B_1\ , \ldots, R_m\ B_m$ where C;, and C does not contain subqueries. Then according to Definition 3

$$\langle Q,d\rangle = \{s_Q(\mu) \mid \nu_1 \in \langle R_1,d\rangle,\ldots,\nu_m \in \langle R_m,d\rangle, \mu = {\nu_1}^{B_1} \odot \cdots \odot {\nu_m}^{B_m}, \langle C\mu,d\rangle\} \tag{1}$$

And it is easy to check that in the case of conditions without subqueries $\langle C\mu,d\rangle = C'\mu$ where C' is obtained from C by replacing true by $\top$, false by $\bot$, and by $\wedge$, or by $\vee$ and not by $\neg$. Analogously in ERA the query can expressed as:

$$ERA(Q,d) = \prod_{e_1 \mapsto Q.A_1,\ldots,e_n \mapsto Q.A_1} \sigma_{C'}(\rho_{B_1}(ERA(R_1,d)) \times \cdots \times \rho_{B_m}(ERA(R_m,d))) \tag{2}$$

with C' defined as above. Now assume that $\eta \in \langle Q,d\rangle$ with cardinality k. This means that there exists $\mu_1,\ldots,\mu_k$ such that $\eta = s_Q(\mu_i)$ for $i = 1\ldots k$. According to 1 this implies that

$$\mu_i = {\nu_1^i}^{B_1} \odot \cdots \odot {\nu_m^i}^{B_m}, \text{ with } \nu_j^i \in \langle R_j,d\rangle \text{ for } j = 1\ldots m \tag{3}$$

Then applying the induction hypothesis to 3 we have that $\lessdot \nu_j^i \gtrdot \in ERA(R_j,d)$ with the same cardinality for $i = 1\ldots n, j = 1\ldots m$, which means that $\rho {\nu_j^i}^{B_j} \in \rho_{B_j}(ERA(R_j,d))$ for $i = 1\ldots n, j = 1\ldots m$, and therefore $\rho\mu_i \in (\rho_{B_1}(ERA(R_1,d)) \times \cdots \times \rho_{B_m}(ERA(R_m,d)))$. From 1 we have that $\langle C\mu,d\rangle = C'\mu_i$ holds, which means that $\rho\mu_i \in \sigma_{C'}(\rho_{B_1}(ERA(R_1,d)) \times \cdots \times \rho_{B_m}(ERA(R_m,d)))$. Finally, taking into account that $\eta = s_Q(\mu_i)$ for $i = 1\ldots k$, and considering that $\lessdot \eta \gtrdot = \lessdot s_Q(\mu_i) \gtrdot$ = $\{Q.A_1 \mapsto (e_1 \lessdot \mu_i \gtrdot), \ldots, Q.A_n \mapsto (e_n \lessdot \mu_i \gtrdot)\}$ = $\prod_{e_1 \mapsto Q.A_1,\ldots,e_n \mapsto Q.A_1}(\lessdot \mu_i \gtrdot)$ we have from 2 that $\lessdot \eta \gtrdot \in ERA(Q,d)$.

□

3 Generating Constraints

In our tool, initially the user chooses a view V and the number of rows of the initial *symbolic database* instance (that is the number of rows of the tables involved in the computation of V). Each attribute value in each row corresponds to a fresh logic variable with its associated domain integrity constraints. The tool tries to obtain a positive test-case by binding the logic variables in the symbolic database. Notice that the number of rows directly affects the result since for some queries does not exists positive test-cases with an specific size. In the future we plan to decide automatically the more convenient size by inspecting the relations definition. Observe also that currently only integer domains are supported. Extending the proposal to other domains such as strings does not seem difficult to achieve, but is beyond the scope of this work.

For instance, in Example 1, suppose that the user decides to look for a positive test-case with two rows in each table. Then, initially the prototype builds the following symbolic instance:

player
id
player.id.0
player.id.1

board		
id	x	y
board.id.0	board.x.0	board.y.0
board.id.1	board.x.1	board.y.1

Notice that player.id.0, player.id.1, board.id.0, board.id.1, board.x.0, board.x.1, board.y.0, board.y.1 represent logic variables. The next definition is employed by the tool to establish the constraints on these variables.

Definition 4 Let D be a database schema and d a database instance. We define $\theta(R,d)$ for every relation R in D as a multiset of pairs (ψ,u) with ψ a first order formula, and u a row. This multiset is defined as follows:

1. For every table T in D such that $d(T) = \{|\mu_1,\ldots,\mu_n|\}$: where $\mu_i = (T.C_1 \mapsto X_{i1},\ldots,T.C_m \mapsto X_{im})$ then:
   - If the definition of T has neither primary key nor foreign key constraints: $\theta(T,d) = \{|(true,\mu_1),\ldots,(true,\mu_n)|\}$.
   - If the definition of T contains primary or foreign key constraints:
     - if T has a primary key constraint for the columns $C_1,\ldots,C_p$: Let T' be the table T without that primary key constraint. Assume that $\theta(T',d) = \{|(\psi_1,\mu_1),\ \ldots,\ (\psi_n,\mu_n)|\}$, then:

$$\theta(T,d) = \{|((\psi_i \wedge (\bigwedge_{j=1,j\neq i}^{n} (\bigvee_{k=1}^{p} \mu_i(T.C_k) \neq \mu_j(T.C_k)))),\mu_i)|i \in 1,\ldots,n|\}$$

     - if T has a foreign key constraint for the columns $T.C_1,\ldots,T.C_f$ referring the columns $T2.C'_1,...T2.C'_f$ from table $T2$: Let T' be the T without that foreign key constraint. Assume that $\theta(T',d) = \{|(\psi_1,\mu_1),\ldots,(\psi_n,\mu_n)|\}$, and that $d(T_2) = \{|\nu_1,\ldots,\nu_{n'}|\}$. Then:

$$\theta(T,d) = \{|((\psi_i \wedge (\bigvee_{j=1,j\neq i}^{n'} (\bigwedge_{k=1}^{f} \mu_i(T.C_k) = \nu_j(T2.C'_k)))),\mu_i)|i \in 1,\ldots,n|\}$$

2. For every view $V =$ create view V($E_1,\ldots,E_n$) as Q,

$$\theta(V,d) = \theta(Q,d)\{V.E_1 \mapsto Q.A_1,\ldots,V.E_n \mapsto A_n\}$$

3. If Q is a basic query of the form:

$$\text{select } e_1,\ldots,e_n \text{ from } R_1\ B_1,\ldots,R_m\ B_m \text{ where } C_w;$$

   Then:

$$\theta(Q,d) = \{|(\psi_1 \wedge \cdots \wedge \psi_m \wedge \varphi(C_w\mu,d), s_Q(\mu)) \mid (\psi_1,\nu_1) \in \theta(R_1,d),\ldots,(\psi_m,\nu_m) \in \theta(R_m,d), \mu = {\nu_1}^{B_1} \odot \cdots \odot {\nu_m}^{B_m}|\}$$

   where

   - $s_Q(\mu) = \{Q.A_1 \mapsto (e_1\mu),\ldots,Q.A_n \mapsto (e_n\mu)\}$,
   - The first order formula $\varphi(C,d)$ is defined as

- If $C \equiv$ false then $\varphi(C,d) = \bot$
- If $C \equiv$ true then $\varphi(C,d) = \top$
- If $C \equiv e$, with e an arithmetic expression involving constants then $\varphi(C,d) = e$
- If $C \equiv e_1 \diamond e_2$, with $\diamond$ a relational operator, then $\varphi(C,d) = (\varphi(e_1,d) \diamond \varphi(e_2,d))$.
- If $C \equiv C_1$ and C_2 then $\varphi(C,d) = \varphi(C_1,d) \wedge \varphi(C_2,d)$
- If $C \equiv C_1$ or C_2 then $\varphi(C,d) = \varphi(C_1,d) \vee \varphi(C_2,d)$
- If $C \equiv$ not C_1 then $\varphi(C,d) = \neg\varphi(C,d)$
- If $C \equiv$ exists Q then suppose that $\theta(Q,d) = \{|(\psi_1,\mu_1),\ldots(\psi_p,\mu_p)|\}$. Then $\varphi(C,d) = (\vee_{j=1}^{p} \psi_j)$.

4. For set queries:

   - $\theta(V_1$ union $V_2,d) = \theta(V_1,d) \cup \theta(V_2,d)$ with $\cup$ the multiset union.
   - $(\psi,\mu) \in \theta(V_1$ intersection $V_2,d)$ with cardinality k iff $(\psi_1,\mu) \in \theta(V_1,d)$ with cardinality k_1, $(\psi_2,\mu) \in \theta(V_2,d)$ with cardinality k_2, $k = min(k_1,k_2)$ and $\psi = \psi_1 \wedge \psi_2$.

Observe that the notation $s_Q(x)$ with Q a query is a shorthand for the row μ with domain $\{E_1,\ldots,E_n\}$ such that $(E_i)x = (e_i)x$, with $i = 1\ldots n$, with select e_1 E_1, $\ldots$, e_n E_n the select clause of Q. If E_i's are omitted in the query, it is assumed that $E_i = e_i$.

The following result and its corollary represent the main result of this paper, stating the soundness and completeness of our proposal:

Theorem 2 *Let D be a database schema and d a valid database instance. Let R be either a relation in D or a query defined using relations in D. Then $\eta \in \langle R,d\rangle$ with cardinality k iff $(true,\eta) \in \theta(R,d)$ with cardinality k.*

Proof. The result is proven by using complete induction on the depth of the syntactic dependence tree defining R.
- If R is a table. Suppose that $d(R) = \{|\mu_1,\ldots,\mu_n|\}$. We distinguish cases depending on the database constraints in the definition of R.

we prove the result using induction on the number of constraints m in the definition of table R.
- If $m = 0$, that is, R is defined with neither primary key nor foreign key constraints. In SOS by Definition 3, $\langle T,d\rangle = d(T) = \{|\mu_1,\ldots,\mu_n|\}$. From Definition 4, $\theta(T,d) = \{|(true,\mu_1),\ldots,(true,\mu_n)|\}$. Then every element η occurs in $\langle R,d\rangle$ with the same cardinality a $(true,\eta)$ in $\theta(R,d)$.
- If $m > 0$ R is defined using at least one primary or foreign key constraints. Select any constraint. Suppose that the selected constraint is a primary key defined by columns $C_1,\ldots,C_p$. Then if $\mu_i \in \langle R,d\rangle$, it appears only once due to the primary key constraints. Let us see that $\mu_i \in \langle R,d\rangle$ iff $(true,\mu_i) \in \theta(R,d)$ (with cardinality 1). From Definition 2.1, $\mu_i \in \langle R,d\rangle$, with d a valid instance implies that

$$\bigwedge_{j=1,j\neq i}^{n} \left(\bigvee_{k=1}^{p} \mu_i(R.C_k) \neq \mu_j(R.C_k)\right) \tag{4}$$

Defined T' as R without this constraint, as explained in Definition 4.1. Then d is also a valid instance for T', and thus $d(T') = d(R)$ which means $\langle T',d\rangle = \langle (R,d)\rangle$. Assume that $\theta(T',d) =$

$\{|(\psi_1,\mu_1),\ldots,(\psi_n,\mu_n)|\}$, and from Definition 4.1

$$\theta(T,d) = \{|(\psi_i \wedge (\bigwedge_{j=1,j\neq i}^{n} (\bigvee_{k=1}^{p} \mu_i(R.C_k) \neq \mu_j(R.C_k)))),\mu_i)|i \in 1,\ldots,n|\} \tag{5}$$

$\mu_i \in \langle R,d\rangle$ iff (from $\langle T',d\rangle = \langle (R,d)\rangle$) $\mu_i \in \langle T',d\rangle$ iff $(true,\mu_i) \in \theta(T',d)$ (by induction hypothesis) iff $\psi_i = true$. Moreover, $\mu_i \in \langle R,d\rangle$ iff 4 is $\top$ (because d is valid) iff (true,μ_i) $\in \theta(R,d)$ (replacing ψ_i and 4 by $\top$ in 5). The result is analogous if the selected constraint is a foreign key.

- If R is a view then $\theta(V,d) = \theta(Q,d)\{V.E_1 \mapsto Q.A_1,\ldots,V.E_n \mapsto Q.A_n\}$ with $E_1,\ldots,E_n$ the attribute names defined by the view (Def. 4), and $\langle V,d\rangle = \langle Q,d\rangle\{V.E_1 \mapsto Q.A_1,\ldots,V.E_n \mapsto Q.A_n\}$ (Def. 3) and the result is straightforward from the induction hypothesis applied to Q.
- If R is a simple query of the form:

  select $e_1,\ldots,e_n$ from $R_1\ B_1\ ,\ldots,R_m\ B_m$ where C;

  According to Definition 3

$$\langle Q,d\rangle = \{s_Q(\mu) \mid \nu_1 \in \langle R_1,d\rangle,\ldots,\nu_m \in \langle R_m,d\rangle, \mu = {\nu_1}^{B_1} \odot \cdots \odot {\nu_m}^{B_m}, \langle C\mu,d\rangle\} \tag{6}$$

and by Definition 4

$$\begin{aligned}\theta(Q,d) = \quad & \{|(\psi_1 \wedge \cdots \wedge \psi_m \wedge \varphi(C\mu,d), s_Q(\mu)) \mid \\ & (\psi_1,\nu_1) \in \theta(R_1,d),\ldots,(\psi_m,\nu_m) \in \theta(R_m,d), \mu = {\nu_1}^{B_1} \odot \cdots \odot {\nu_m}^{B_m}|\}\end{aligned} \tag{7}$$

The first step is to check that

$$\varphi(C\mu,d) \text{ iff } \langle C\mu,d\rangle \tag{8}$$

The result is straightforward from the respective definitions (3 and 4) except in the case of existential subqueries. In this case Definition 3 indicates that an existential subquery Q' is transformed into $\langle Q',d\rangle \neq \emptyset$, which is true iff there is some $\eta \in \langle Q',d\rangle$. By the inductive hypothesis this means that $(true,\eta) \in \theta(Q',d)$. Thus, in the definition 4 the multiset $\theta(Q',d) = \{|(\psi_1,\mu_1),\ldots(\psi_p,\mu_p)|\}$ contains at least one tuple (ψ_i,μ_i) with $\psi_i = true$ for some $1 \leq i \leq p$, which implies that $\varphi(C\mu,d) = (\vee_{j=1}^{p} \psi_j)$ is true, proving 8.

Then $(true,\eta) \in \theta(Q,d)$ with cardinality k iff there are exactly k values such that for $i = 1\ldots k$

$$\mu_i = {\nu_1^i}^{B_1} \odot \cdots \odot {\nu_m^i}^{B_m}, \text{ for some } (\psi_1^i,\nu_1^i) \in \theta(R_1,d),\ldots,(\psi_m^i,\nu_m^i) \in \theta(R_m,d) \tag{9}$$

$$\eta = s_Q(\mu_i) \tag{10}$$

$$\psi_1^i = true,\ldots,\psi_m^i = true \tag{11}$$

$$\varphi(C\mu,d) = true \text{ and thus by Def. 3 } \langle C\mu,d\rangle = true \tag{12}$$

From 11 and considering 9, applying the induction hypothesis:

$$\nu_1^i \in \langle R_1,d\rangle,\ldots,\nu_m^i \in \langle R_m,d\rangle, i = 1\ldots k \tag{13}$$

and thus $s_Q(\mu_i) \in \langle Q,d\rangle$ in 6 for $i = 1 \ldots m$. That is (considering 10), $\eta \in \langle Q,d\rangle$ with multiplicity k.

- If R is a query of the form: Q_1 union Q_2 then $\langle Q,d\rangle = \langle Q1,d\rangle \cup \langle Q2,d\rangle$ (Def. 3), $\theta(V_1 \text{ union } V_2,d) = \theta(V_1,d) \cup \theta(V_2,d)$ (Def. 4), and the result is an easy consequence of the induction hypothesis.
- If R is a query of the form: Q_1 intersection Q_2 the result is similar to the previous case and it is omitted for the sake of space. □

Observe that in [CGS10] the proof was restricted to queries without subqueries due to the limitations of the Extended Relational Algebra [GUW08] used as operational semantics.

The following corollary contains the idea for generating constraints that will yield the PTCs:

Corollary 1 *Let D be a database schema and d_s a symbolic database instance. Let R be a relation in D such that $\theta(R) = \{|(\psi_1,\mu_1),\ldots(\psi_n,\mu_n)|\}$, and η a substitution satisfying d_s. Then $d_s\eta$ is a PTC for R iff $(\bigvee_{i=1}^{n} \psi_i)\eta = true$.*

Proof. Straightforward from Theorem 2: $(\bigvee_{i=1}^{n} \psi_i)\eta = true$ iff there is some ψ_i with $1 \leq i \leq n$ such that $\psi_i\eta = true$ iff $(\mu_i\eta) \in \langle R\rangle$ iff $\langle R\rangle \neq \emptyset$. □

4 Prototype

In this section, we comment on some aspects of our implementation and show a system session for our running example. The SQL Test Case Generator (*STCG*) is implemented in the programming language C++.

The input of STCG consists of a table or view name R, and a SQL file containing the definition of R and of all the relations in its syntactic dependence tree. The goal is to obtain a positive test-case for R.

STCG can be seen as a pipeline, which helps to maintain and extend the implementation. In a first phase the prototype generates a representation in C++ of the CSP. This phase consists of two steps:

1. **SQL parser**: STCG includes a SQL parser to pre-process the input file. The result is a C++ representation of the database schema. This parser has been produced using *GNU Bison* [GNU], a general-purpose parser generator, and *Flex* [CF04], a tool for generating scanners.

2. **Formula Generator**: The core of STCG. Takes as input the output of the previous step, and following Definition 4 described in section 3, generates the first order logic formula representing the constraints associated to the table or view specified.

Figure 1 shows a diagram of this first phase. Next, we want to satisfy this formula by binding the logic variables, thus obtaining a positive test case if possible.

In the second phase (see Figure 2) the logic formula is translated into a format accepted by some external constraint solver. We have used *G12/CPX* as solver, which is distributed with *G12 MiniZinc distribution*, however, any solver that implements FlatZinc can be used for this purpose. *MiniZinc* [NSB+07] is a medium-level constraint modeling language. We translate our

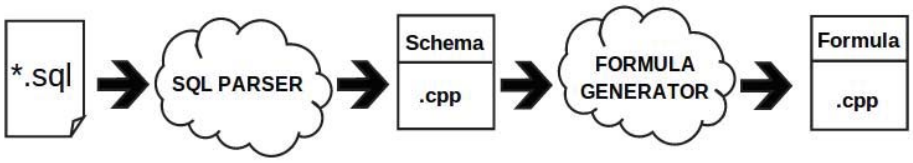


Figure 1: Pipeline of STCG.

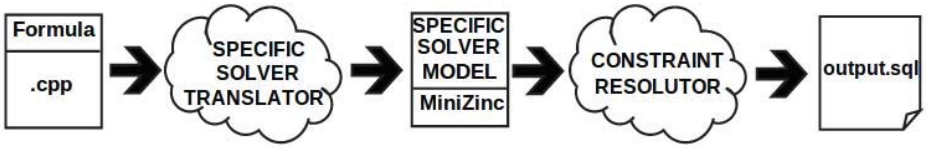


Figure 2: Pipeline of STCG extended with an input builder for a solver.

logic formula into a *MiniZinc model*, so it can be solved, returning, if possible, the positive test-cases. Other constraint programing languages can be added by implementing new translators.

Below we present a system session for the example at hand. Consider that the input SQL file corresponds to Example 1.

Following the pipeline, the SQL parser takes this file as input and generates an internal C++ representation of the database schema (it can be obtained at the command line using the "-v" option of STCG). Suppose that the user intends to generate a test case for view *checked* indicating that tables *player* and *board* must have two rows at most. Thus, the *Formula Generator* module takes the database schema as input and generates the following formula (where symbolic variables are of the form *table name.column name.row*):

```
#(checked)=
 {(((((board.id.0 ==  player.id.0) OR (board.id.1 == player.id.0))
 AND ...
 (not (((((board.x.0 - board.x.1) * (board.y.0 - board.y.1)) == 0)
 AND (board.id.1 != board.id.0)) OR ...)))))),
 {checked.id -> player.id.0}), ...}
```

This corresponds to $\theta(checked, board)$ without considering primary and foreign keys (displayed below). For the sake of space we display only part of the first formula and the first row $\{\mathsf{checked.id} \rightarrow \mathsf{player.id.0}\}$. This row indicates that all the pieces of player.id.0 are checked if the formula holds. Firstly, the formula indicates that player.id.0 must have a piece on either the position corresponding to board.id.0 or to board.id.1. The other part of the formula displayed indicates that there is a check between positions contained in board.x.0,board.y.0, and board.x.1,board.y.1 for two different players.

Primary key constraint of table *player* is shown below. For simplicity, only logic formula for the first row $\{$player.id $\rightarrow$ player.id.0$\}$ is displayed. That formula indicates that this table cannot have two players with the same id attribute, that is player.id.0 != player.id.1.

```
#(player)={(player.id.0 != player.id.1),
{player.id -> player.id.0}), ...}
```

Next, we display primary key and foreign key constraints of *board* for the first row $\{$board.x $\rightarrow$ board.x.0, board.y $\rightarrow$ board.y.0, board.id $\rightarrow$ board.id.0$\}$. That formula indicates that id attribute of each row of this table must reference one of id attributes of the table *player* (foreign key), that is board.id.0 == player.id.0 $\wedge$ board.id.1 == player.id.0, and two pieces cannot be in the same game board position, that is (board.x.0 != board.x.1) $\vee$ (board.y.0 != board.y.1) (primary key).

```
#(board)=
{(((board.id.0 == player.id.0) AND (board.id.1 == player.id.0))
AND ((board.x.0 != board.x.1) OR (board.y.0 != board.y.1)),
{board.x-> board.x.0,board.y-> board.y.0,board.id-> board.id.0}),...}
```

Actually, STCG generates only one formula, with primary key and foreign key constraints included, but we show them separated for simplicity.

The next step of the pipeline is to solve this constraints if possible. As mentioned above, we use the MiniZinc constraint modeling language for this purpose. We introduce a new step to the pipeline, that will translate our *Formula* to a *MiniZinc model*. This model consists of four sections: variable declarations, constraint specification, solver invocation and output formatting.

The model is saved in a file that can be later executed by MiniZinc, returning the positive test-case if possible. To obtain meaningful results we restrict the domain of symbolic variables to 1..5 in this session. In the future, our tool will support domain constraints of symbolic variables. The result returned by MiniZinc is:

```
INSERT INTO board(id,x,y) VALUES (1,1,1);
INSERT INTO board(id,x,y) VALUES (2,1,5);
INSERT INTO player(id) VALUES(1);
INSERT INTO player(id) VALUES(2);
```

This result is a positive test-case, written as a set of SQL insert statements that populates the tables of the database with the following data:

player
id
1
2

board		
id	x	y
1	1	1
2	1	5

Thus, the board has player 1 in position (1,1) and player 2 in position (1,5) and both players are checked as defined in the example 1. We can check that this instance is a positive test-case by executing the SQL query select * from checked, which returns the following non empty result:

id
1
2

5 Conclusions and Future Work

We have presented a technique for generating positive test-cases for sets of correlated SQL views. Our setting considers that a database instance constitutes a test-case for a view if the view computes at least one tuple with respect to the instance. Our proposal is to reduce the problem of obtaining a test-case to a Constraint Satisfaction Problem over finite domains. The generated constraints take into account primary and foreign key database constraints if they are defined in an SQL table definition statement. The work improves the proposal of [CGS10] by introducing a complete treatment of existential subqueries. In order to prove the correctness of the technique, a new operational semantics for SQL views without aggregates is introduced. The theoretical ideas have been implemented in a working prototype called STCG, available at:

https://gpd.sip.ucm.es/trac/gpd/wiki/GpdSystems/STCG

The prototype takes a set of views and the name of a particular view V, and generates a positive test-case for V. It uses MiniZinc [NSB+07] as constraint modeling language. The output is given in the form of a list of SQL insert commands that create the database instance.

Although the framework presented is useful for many types of queries, including those defined by set operations and those queries including existential subqueries, it is only a first step since there are many useful SQL features still not supported by our tool. As immediate future work, we plan the integration of aggregate subqueries. The theoretical problem was already addressed in [CGS10], and the implementation seems straightforward. It would also be interesting to extend the SQL operational semantics to aggregates in order to prove the soundness of the proposal. Another limitation that must be solved is the inclusion of null values, which is currently not supported. However, the main limitation of the tool, and thus the main goal for our future work, is to extend the data types allowed for SQL columns. Currently the type of table attributes is limited to integer, and at least varchar (that is, string types) are required in order to obtain a really applicable tool.

From the theoretical point of view it would be interesting to define the introduction of existential nested subqueries directly into ERA, thus replacing SOS by an enriched version of ERA. Another theoretical improvement would be to determine formally the minimum number of rows necessary for ensuring that a positive test-case exists. Currently, it is the user who indicates the number of rows for each table, but we think that the size could be determined by automatically examining the SQL code.

Bibliography

[AO08] P. Ammann, J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[CF04] J. Coelho, M. Florido. CLP(Flex): Constraint Logic Programming Applied to XML Processing. In *Proceedings of the CoopIS/DOA/ODBASE*. Pp. 1098–1112. Springer LNCS 3291, Heidelberg, Germany, 2004.

[CGS10] R. Caballero, Y. García-Ruiz, F. Sáenz-Pérez. Applying Constraint Logic Programming to SQL Test Case Generation. In Blume et al. (eds.), *Functional and Logic Pro-*

gramming. Lecture Notes in Computer Science 6009, pp. 191–206. Springer Berlin / Heidelberg, 2010.

[CT04] M. J. S. Cabal, J. Tuya. Using an SQL coverage measurement for testing database applications. In Taylor and Dwyer (eds.), *SIGSOFT FSE*. Pp. 253–262. ACM, 2004.

[GNU] GNU. Bison - GNU parser generator.

[GUW08] H. Garcia-Molina, J. D. Ullman, J. Widom. *Database Systems: The Complete Book*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2008.

[NSB+07] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, G. Tack. MiniZinc: Towards a Standard CP Modelling Language. In Bessiere (ed.). Lecture Notes in Computer Science 4741, pp. 529–543. Springer, 2007.

[SQL92] SQL, ISO/IEC 9075:1992, third edition. 1992.

[ST09] M. Surez-Cabal, J. Tuya. Structural Coverage Criteria for Testing SQL Queries. *Journal of Universal Computer Science* 15(3):584–619, 2009.

[ZHM97] H. Zhu, P. A. V. Hall, J. H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys* 29:366–427, 1997.

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): Actas de las "XII Jornadas sobre Programación y Lenguajes (PROLE'2012), Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

Debugging Fuzzy XPath Queries

Jesús M. Almendros-Jiménez[1], Alejandro Luna[2] and Ginés Moreno[3]

[1] jalmen@ual.es
Dpto. de Lenguajes y Computación
Universidad de Almería
04120 Almería (Spain)
[2] Alejandro.Luna@alu.uclm.es
[3] Gines.Moreno@uclm.es
Dept. of Computing Systems
University of Castilla-La Mancha
02071 Albacete (Spain)

Abstract: In this paper we report a preliminary work about XPath debugging. We will describe how we can manipulate an XPath expression in order to obtain a set of alternative XPath expressions that match to a given XML document. For each alternative XPath expression we will give a chance degree that represents the degree in which the expression deviates from the initial expression. Thus, our work is focused on providing the programmer a repertoire of paths that (s)he can use to retrieve answers. The approach has been implemented and tested.

Keywords: XPath; Fuzzy (Multi-adjoint) Logic Programming; Debugging

1 Introduction

The eXtensible Markup Language (XML) is widely used in many areas of computer software to represent machine readable data. XML provides a very simple language to represent the structure of data, using tags to label pieces of textual content, and a tree structure to describe the hierarchical content. XML emerged as a solution to data exchange between applications where tags permit to locate the content. XML documents are mainly used in databases. The XPath language [BBC+07] was designed as a query language for XML in which the path of the tree is used to describe the query. XPath expressions can be adorned with boolean conditions on nodes and leaves to restrict the number of answers of the query. XPath is the basis of a more powerful query language (called XQuery) designed to join multiple XML documents and to give format to the answer.

In spite of the simplicity of the XPath language, the programmer usually makes mistakes when (s)he describes the path in which the data are allocated. Tipically, (s)he omits some of the tags of the path, s(he) adds more than necessary, and (s)he also uses similar but wrong tag names. When the query does not match to the tree structure of the XML tree, the answer is empty. However, we can also find the case in which the query matches to the XML tree but the answer does not satisfy the programmer. Due to the inherent flexibility of XML documents, the same tag can occur at several positions, and the programmer could find answers that do not correspond to her(is) expectations. In other words, (s)he finds a correct path, but a wrong answer. We can also

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): PROLE'2012, pp. 119-133, ISBN:978-84-15487-27-2.
Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

consider the case in which a boolean condition is wrong, expressing a wrong range, and several conditions that do not hold at the same time. When the programmer does not find the answer (s)he is looking for, there is a mechanism that (s)he can try to debug the query. In XPath there exists an operator, denoted by '//', that permits to look for the tag from that position. However, it is useless when the tag is present at several positions, since even though the programmer finds answers, (s)he does not know whether they are close to h(er) expectations.

XPath debugging has to take into account the previous considerations. Particularly, there is an underlying notion of *chance degree*. When the programmer makes mistakes, the number of bugs can be higher or lower, and the chance degree is proportional to them. Moreover, there are several ways on which each bug can be solved, and therefore the chance degree is also dependent from the number of solutions for each bug, and the quality of each solution. The quality of a solution describes the number of changes to be made. Finally, there is a case in which we have also focused our work. The case in which the mistake comes from a similar but wrong used tag. Here, the chance degree comes from the semantic closeness of the used tag.

In this paper we report a preliminary work about XPath debugging. We will describe how we can manipulate an XPath expression in order to obtain a set of alternative XPath expressions that match to a given XML document. For each alternative XPath expression we will give a chance degree that represents the degree in which the expression deviates from the initial expression. Thus, our work is focused on providing the programmer a repertoire of paths that (s)he can use to retrieve answers.

We propose that XPath debugging is guided by the programmer that initially establishes a value (a real value between 0 and 1), that the debugger uses to penalize each bug. Each bug found is penalized with such a value, and thus the chance degree is proportional to the value. Additionally, we assume that the debugger is equipped with a table of similarities, that is, a table in which pairs of similar words are assigned to a value between 0 and 1. It makes possible that chance degree is also computed from similarity degrees. The debugger reports a set of annotated paths in an extended XPath syntax in which we have incorporated three annotations: `JUMP`, `SWAP` and `DELETE`. `JUMP` is used to represent that some tags have been added to the original expression, `SWAP` is used to represent that a tag has been changed by another, and `DELETE` is used to represent that a tag has been removed. The reported XPath expressions update the original XPath expression, that is, the case `JUMP` incorporates '//' at the position in which the bug is found, the case `SWAP` includes the new tag, and the case `DELETE` removes the wrong tag.

Additionally, our proposal permits the programmer tests the reported XPath expressions. The annotated XPath expressions can be executed obtaining a ranked set of answers with respect to the chance degree. It facilitates the process of debugging because the programmer can visualize the answers to each query.

The approach has been implemented and tested (see http://dectau.uclm.es/fuzzyXPath/debug_fuzzyXPathTest.php). The implementation has been developed on top of the recently proposed fuzzy XPath extension [ALM12], which uses *fuzzy logic programming* to provide a fuzzy taste to XPath expressions. Although our approach can be applied to standard (crisp) XPath expressions, chance degrees in XPath debugging fits well with our proposed framework. Particularly, XPath debugging annotations can be seen as annotations of XPath expressions similar to the proposed `DEEP` and `DOWN` of [ALM12]. `DEEP` and `DOWN` serve to annotate XPath expressions and to obtain a ranked set of answers depending on they occur, more deeply and from top to down. Each answer

is annotated with a *RSV (Retrieval Status Value)* which describes the degree of satisfaction of the answer. Here JUMP, SWAP and DELETE penalize the answers of annotated XPath expressions. When annotated XPath expressions are executed, we obtain a ranked set of answers with respect to the *CD (Chance Degree)* of the programmer. DEEP and JUMP have, in fact, the same behavior: JUMP proportionally penalizes answers as deep as they occur. Finally, and in order to cover with SWAP, we have incorporated to our framework similarity degrees.

The structure of the paper is as follows. Section 2 will summarize our previous work and will introduce concepts that are closely related to the proposed debugging technique. Section 3 will describe the debugging technique. Section 4 will show some implementation details, and finally, Section 5 will conclude and present future work.

2 Fuzzy XPath

In this section we will summarize the main elements of our proposed fuzzy XPath language described in [ALM12, ALM11].

Our fuzzy XPath incorporates two structural constraints called DOWN and DEEP to which a certain degree of relevance is associated. So, whereas DOWN provides a ranked set of answers depending on the path they are found from "top to down" in the XML document, DEEP provides a ranked set of answers depending on the path they are found from "left to right" in the XML document. Both structural constraints can be used together, assigning degree of importance with respect to the distance to the root XML element.

Secondly, our fuzzy XPath incorporates fuzzy variants of *and* and *or* for XPath conditions. Crisp *and* and *or* operators are used in standard XPath over boolean conditions, and enable to impose boolean requirements on the answers. XPath boolean conditions can be referred to attribute values and node content, in the form of equality and range of literal values, among others. However, the *and* and *or* operators applied to two boolean conditions are not precise enough when the programmer does not give the same value to both conditions. For instance, some answers can be discarded when they could be of interest by the programmer, and accepted when they are not of interest. Besides, programmers would need to know in which sense a solution is better than another. When several boolean conditions are imposed on a query, each one contributes to satisfy the programmer's preferences in a different way and perhaps, the programmer's satisfaction is distinct for each solution.

We have enriched the arsenal of operators of XPath with fuzzy variants of *and* and *or*. Particularly, we have considered three versions of *and*: *and+*, *and*, *and-* (and the same for *or* : *or+*, *or*, *or-*) which make more flexible the composition of fuzzy conditions. Three versions for each operator that come for free from our adaptation of fuzzy logic to the XPath paradigm. One of the most known elements of fuzzy logic is the introduction of fuzzy versions of classical boolean operators. *Product*, *Łukasiewicz* and *Gödel* fuzzy logics are considered as the most prominent logics and give a suitable semantics to fuzzy operators. Our contribution is now to give sense to fuzzy operators into the XPath paradigm, and particularly in programmer's preferences. We claim that in our work the fuzzy versions provide a mechanism to force (and debilitate) conditions in the sense that stronger (and weaker) programmer preferences can be modeled with the use of stronger (and weaker) fuzzy conditions. The combination of fuzzy operators in queries

permits to specify a ranked set of fuzzy conditions according to programmer's requirements.

Furthermore, we have equipped XPath with an additional operator that is also traditional in fuzzy logic: the average operator *avg*. This operator offers the possibility to explicitly give weight to fuzzy conditions. Rating such conditions by *avg*, solutions increase its weight in a proportional way. However, from the point view of the programmer's preferences, it forces the programmer to quantify his(er) wishes which, in some occasions, can be difficult to measure. For this reason, fuzzy versions of *and* and *or* are better choices in some circumstances.

Finally, we have equipped our XPath based query language with a mechanism for thresholding programmer's preferences, in such a way that programmer can request that requirements are satisfied over a certain percentage.

The proposed fuzzy XPath is described by the following syntax:

xpath :=	['['deep-down']']path
path :=	literal | text() | node | @att | node**/**path | node**//**path
node :=	QName | QName[cond]
cond :=	xpath op xpath | xpath num-op number
deep :=	DEEP=number
down :=	DOWN=number
deep-down :=	deep | down | deep '**;**' down
num-op :=	**>** | **=** | **<** | **<>**
fuzzy-op :=	**and** | **and+** | **and-** | **or** | **or+** | **or-** | **avg** | **avg**{number,number}
op :=	num-op | fuzzy-op

Basically, our proposal extends XPath as follows:

- **Structural constraints**. A given XPath expression can be adorned with «[DEEP = r_1; DOWN = r_2]» which means that the *deepness* of elements is penalized by r_1 and that the *order* of elements is penalized by r_2, and such penalization is proportional to the distance (i.e., the length of the branch and the weight of the tree, respectively). In particular, «[DEEP = 1; DOWN = r_2]» can be used for penalizing only w.r.t. document order. DEEP works for //, that is, the deepness in the XML tree is only computed when descendant nodes are explored, while DOWN works for both / and //. Let us remark that DEEP and DOWN can be used several times on the main *path* expression and/or any other *sub-path* included in conditions.

- **Flexible operators in conditions**. We consider three fuzzy versions for each one of the classical conjunction and disjunction operators (also called connectives or aggregators) describing *pessimistic*, *realistic* and *optimistic* scenarios, see Figure 1. In XPath expressions the fuzzy versions of the connectives make harder to hold boolean conditions, and therefore can be used to debilitate/force boolean conditions. Furthermore, assuming two given *RSV*'s r_1 and r_2, the *avg* operator is obviously defined with a fuzzy taste as $(r_1 + r_2)/2$, whereas its *priority-based* variant, i.e. $avg\{p_1, p_2\}$, is defined as $(p_1 * r_1 + p_2 * r_2)/p_1 + p_2$.

In general, a fuzzy XPath expression defines, w.r.t. an XML document, a sequence of subtrees of the XML document where each subtree has an associated RSV. XPath conditions, which are defined as fuzzy operators applied to XPath expressions, compute a new RSV from the RSVs of the involved XPath expressions, which at the same time, provides a RSV to the node. In order to

Figure 1: Fuzzy Logical Operators

$$
\begin{array}{lcllll}
\&_{\mathrm{P}}(x,y) & = & x*y & |_{\mathrm{P}}(x,y) & = & x+y-x*y \quad \textit{Product: and/or} \\
\&_{\mathrm{G}}(x,y) & = & \min(x,y) & |_{\mathrm{G}}(x,y) & = & \max(x,y) \quad \textit{Gödel: and+/or-} \\
\&_{\mathrm{L}}(x,y) & = & \max(x+y-1,0) & |_{\mathrm{L}}(x,y) & = & \min(x+y,1) \quad \textit{Łuka.: and-/or+}
\end{array}
$$

Figure 2: XML skeleton represented as a tree

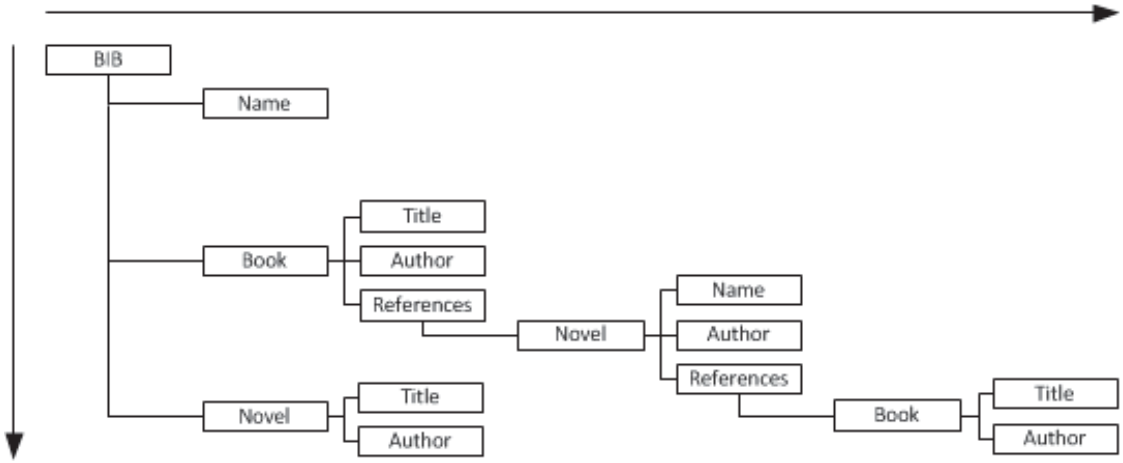


Figure 3: Input XML document in our examples

```
<bib>
    <name>Classic Literature</name>
    <book year="2001" price="45.95">
        <title>Don Quijote de la Mancha</title>
        <author>Miguel de Cervantes Saavedra</author>
        <references>
            <novel year="1997" price="35.99">
                <name>La Galatea</name>
                <author>Miguel de Cervantes Saavedra</author>
                <references>
                    <book year="1994" price="25.99">
                        <title>Los trabajos de Persiles y Sigismunda</title>
                        <author>Miguel de Cervantes Saavedra</author>
                    </book>
                </references>
            </novel>
        </references>
    </book>
    <novel year="1999" price="25.65">
        <title>La Celestina</title>
        <author>Fernando de Rojas</author>
    </novel>
</bib>
```

illustrate these explanations, let us see some examples of our proposed fuzzy version of XPath according to the XML document shown of Figure 3, whose *skeleton* is depicted in Figure 2.

Figure 4: Execution of the query «/bib[DEEP=0.8;DOWN=0.9]//title»

Document	RSV computation

`<result>`
`  <title rsv="0.8000">Don Quijote de la Mancha</title>`
`  <title rsv="0.7200">La Celestina</title>`
`  <title rsv="0.2949">Los trabajos de Persiles y Sigismunda</title>`
`</result>`
	$0.8000 = 0.8$
	$0.7200 = 0.8 * 0.9$
	$0.2949 = 0.8^5 * 0.9$

Figure 5: Execution of the query «//book[@year<2000 avg{3,1} @price<50]/title»

Document	RSV computation

`<result>`
`  <title rsv="1.00">Los trabajos de Persiles y Sigismunda</title>`
`  <title rsv="0.25">Don Quijote de la Mancha</title>`
`</result>`
	$1.00 = (3*1+1*1)/(3+1)$
	$0.25 = (3*0+1*1)/(3+1)$

Figure 6: Execution of the query «/bib[DEEP=0.5]//book[@year<2000 avg{3,1} @price<50]/title»

Document	RSV computation

`<result>`
`  <title rsv="0.25">Don Quijote de la Mancha</title>`
`  <title rsv="0.0625">Los trabajos de Persiles y ..</title>`
`</result>`
	$0.25 = (3*0+1*1)/(3+1)$
	$0.0625 = 0.5^4 * (3*1+1*1)/(3+1)$

Example 1 *Let us consider the fuzzy XPath query of Figure 4 requesting title's penalizing the occurrences from the document root by a proportion of* 0.8 *and* 0.9 *by nesting and ordering, respectively, and for which we obtain the file listed in Figure 4. In such document we have included as attribute of each subtree, its corresponding RSV. The highest RSVs correspond to the main books of the document, and the lowest RSVs represent the books occurring in nested positions (those annotated as related references).*

Example 2 *Figure 5 shows the answer associated to a search of books, possibly referenced* directly or indirectly *from other books, whose publishing year and price are relevant but the year is three times more important than the price. Finally, in Figure 6 we combine both kinds of (structural/conditional) operators, and the ranked list of solutions is reversed, where "Don Quijote" is not penalized with* DEEP.

3 Debugging XPath

In this section we propose a debugging technique for XPath expressions. Our debugging process accepts as inputs a query Q preceded by the $[\texttt{DEBUG} = r]$ command, where r is a real number in the unit interval. For instance, «[DEBUG=0.5]/bib/book/title». Assuming an input XML document like the one depicted in Figures 2 and 3, the debugging produces a set of alternative queries $Q_1, ..., Q_n$ packed into an output XML document, like the one shown in Figure 7. The document

has the following structure:

<result>
<query cd="r_1" $attributes_1$> Q_1 </query>
...
<query cd="r_n" $attributes_n$> Q_n </query>
</result>

where the set of alternatives is ordered with respect to the CD key. This value measures the chance degree of the original query with respect to the new one, in the sense that as much changes are performed on Q_i and as more *traumatic* they are with respect to Q, then the CD value becomes lower.

Figure 7: Debugging of the query «[DEBUG=0.5]/bib/book/title»

```
<result>
  <query cd="1.0">/bib/book/title</query>
  <query cd="0.8" book="novel">/bib/[SWAP=0.8]novel/title</query>
  <query cd="0.5" book="//">/bib/[JUMP=0.5]//title</query>
  <query cd="0.5" bib="//">/[JUMP=0.5]//book/title</query>
  <query cd="0.45" book="" title="name">/bib/[DELETE=0.5][SWAP=0.9]name</query>
  <query cd="0.4" bib="//" book="novel">/[JUMP=0.5]//[SWAP=0.8]novel/title</query>
  <query cd="0.25" book="" title="//">/bib/[DELETE=0.5][JUMP=0.5]//title</query>
  <query cd="0.25" book="//" book="">/bib/[JUMP=0.5]//[DELETE=0.5]title</query>
  <query cd="0.25" bib="" book="//">/[DELETE=0.5][JUMP=0.5]//book/title</query>
  <query cd="0.25" bib="//" book="//">/[JUMP=0.5]//[JUMP=0.5]//title</query>
  <query cd="0.25" bib="//" bib="">/[JUMP=0.5]//[DELETE=0.5]book/title</query>
  <query cd="0.225" title="//" title="//" title="name">/bib/book/[JUMP=0.5]//[JUMP
      =0.5]//[SWAP=0.9]name</query>
  <query cd="0.225" bib="" book="//" title="name">/[DELETE=0.5][JUMP=0.5]//[SWAP=0.9]
      name</query>
  <query cd="0.225" bib="//" book="" title="name">/[JUMP=0.5]//[DELETE=0.5][SWAP=0.9]
      name</query>
  <query cd="0.2" bib="" book="//" book="novel">/[DELETE=0.5][JUMP=0.5]//[SWAP=0.8]
      novel/title</query>
   .........
</result>
```

In Figure 7, the first alternative, with the highest CD, is just the original query, thus, the CD is *1*, whose further execution with fuzzy XPath returns «Don Quijote de La Mancha». As was commented in the introduction, we have assumed the debugger is ran even when the set of answers is not empty, like in this case. The remaining options give different CD's depending on the chance degree, and provide XPath expressions annotated with JUMP, DELETE and SWAP.

In order to explain the way in which our technique generates the attributes and content of each *query* tag in the output XML document, let us consider a generic path Q of the form: «$[\texttt{DEBUG} = r]/tag_1/.../tag_i/tag_{i+1}/...$», where we say that tag_i is at level i in the original query. So, assume that during the exploration of the input query Q and the XML document D, we find that tag_i in Q does not occurs at level i in (a branch of) D. Then, we consider the following three situations:

Swapping case: Instead of tag_i, we find tag'_i at level i in the input XML document D, being tag_i and tag'_i two similar terms with similarity degree s. Then, we generate an alternative query by adding the attribute tag_i ="tag'_i" and replacing in the original path the occurrence "$tag_i/$" by "$[\texttt{SWAP} = s]tag'_i/$".
The second query proposed in Figure 7 illustrates this case:

« <query cd="0.8" book="novel">/bib/[SWAP=0.8]novel/title</query> ».

Let us observe that : 1) we have included the attribute «book="novel"» in order to suggest that instead of looking now for a *book*, finding a *novel* should be also a good alternative, 2) in the path we have replaced the tag *book* by *novel* and we have appropriately annotated the exact place where the change has been performed with the annotation *[SWAP=0.8]* and 3) the CD of the new query has been adjusted with the *similarity degree* 0.8 of the exchanged tags.

Now, we can run the (fuzzy) XPath queries «/bib/novel/title» and even «/bib/[SWAP=0.8]novel-/title» (see Figure 8). In both cases we obtain the same result, i.e., «La Celestina», but with different RSVs (1 and 0.8).

Figure 8: Execution of the query «/bib/[SWAP=0.8]novel/title»

```
<result>
  <title rsv="0.8">La Celestina</title>
</result>
```

Jumping case: Even when tag_i is not found at level i in the input XML document D, tag_{i+1} appears at a deeper level (i.e., greater than i) in a branch of D. Then, we generate an alternative query by adding the attribute tag_i=*"//"*, which means that tag_i has been jumped, and replacing in the path the occurrence *"tag_i/"* by *"[JUMP=r]//"*, being r the value associated to DEBUG.

Figure 9: Execution of the query «/bib/[JUMP=0.5]//title»

```
<result>
  <title rsv="0.5">Don Quijote de la Mancha</title>
  <title rsv="0.5">La Celestina</title>
  <title rsv="0.03125">Los trabajos de Persiles y Sigismunda</title>
</result>
```

Figure 10: Execution of the query «/[JUMP=0.5]//book/title»

```
<result>
  <title rsv="0.5">Don Quijote de la Mancha</title>
  <title rsv="0.03125">Los trabajos de Persiles y Sigismunda</title>
</result>
```

This situation is illustrated by the third and fourth queries in Figure 7, where we propose to jump the tags *book* and *bib*. The execution of the queries returns different results, as shown in

Figures 9 and 10, where JUMP produces similar effects to the DEEP command explained in the previous section, that is, as more tags are jumped their resulting CD's become lower.

Deletion case: This situation emerges when at level i in the input XML document D, we found tag_{i+1} instead of tag_i. So, the intuition tell us that tag_i should be removed from the original query Q and hence, we generate an alternative query by adding the attribute tag_i=*""* and replacing in the path the occurrence *"tag_i/"* by *"[*DELETE*=r]"*, being r the value associated to DEBUG.

This situation is illustrated by the fifth query in Figure 7, where the deletion of the tag *book* is followed by a swapping of similar tags *title* and *name*. The CD *0.45* associated to this query is defined as the *product* of the values associated to both DELETE (*0.5*) and SWAP (*0.9*), and hence the chance degree of the original one is lower than the previous examples.

As seen in Figure 11, the execution of our new query is able to retrieve the information contained in the first branch of the input XML document listed in Figures 2 and 3. Here we illustrate that execution of debugged XPath expressions reveals hidden answers that can fulfill the programmer expectations.

Figure 11: Execution of the query «/bib/[DELETE=0.5][SWAP=0.9]name»

```
<result>
  <name rsv="0.45">Classic Literature</name>
</result>
```

As we have seen in the previous example, the combined use of one or more debugging commands (SWAP, JUMP and DELETE) is not only allowed but also frequent. In other words, it is possible to find several debugging points.

In Figure 12, we can see the execution of the query:

« <query cd="0.225" bib="" book="//" title="name">/[DELETE=0.5][JUMP=0.5]// [SWAP=0.9]name</query> »

The CD *0.225* is quite low, and therefore the chance degree is low, since it has been obtained by multiplying the three values associated to the deletion of the tag *bib* (*0.5*), jumping the tag *book* (*0.5*) and the swapping of *title* by *name* (*0.9*).

Figure 12: Execution of the query «/[DELETE=0.5][JUMP=0.5]//[SWAP=0.9]name»

```
<result>
  <name rsv="0.225">Classic Literature</name>
  <name rsv="0.028125">La Galatea</name>
</result>
```

The wide range of alternatives (Figure 7 is still incomplete), reveals the flexibility of our technique. The programmer is free to use the alternative queries to execute them, and to inspect results up to the expectations are covered.

Finally, we would like to remark that even when we have worked with a very simple query with three tags in our examples, our technique works with more complex queries with large paths

and connectives in boolean conditions, as well as DEBUG used in several places on the query.

For instance, in Figure 13 (compare it with Figure 7) we show the result of debugging the following query: «[DEBUG=0.7]/bib/[DEBUG=0.6]book/[DEBUG=0.5]title».

Figure 13: Debugging of the query «[DEBUG=0.7]/bib/[DEBUG=0.6]book/[DEBUG=0.5]title»

```
<result>
  <query cd="1.0">/bib/book/title</query>
  <query cd="0.8" book="novel">/bib/[SWAP=0.8]novel/title</query>
  <query cd="0.7" bib="//">/[JUMP=0.7]//book/title</query>
  <query cd="0.6" book="//">/bib/[JUMP=0.6]//title</query>
  <query cd="0.56" bib="//" book="novel">/[JUMP=0.7]//[SWAP=0.8]novel/title</query>
  <query cd="0.54" book="" title="name">/bib/[DELETE=0.6][SWAP=0.9]name</query>
  <query cd="0.42" bib="" book="//">/[DELETE=0.7][JUMP=0.6]//book/title</query>
  <query cd="0.42" bib="//" book="//">/[JUMP=0.7]//[JUMP=0.6]//title</query>
  <query cd="0.378" bib="" book="//" title="name">
                    /[DELETE=0.7][JUMP=0.6]//[SWAP=0.9]name</query>
  <query cd="0.378" bib="//" book="" title="name">
                    /[JUMP=0.7]//[DELETE=0.6][SWAP=0.9]name</query>
  <query cd="0.336" bib="" book="//" book="novel">
                    /[DELETE=0.7][JUMP=0.6]//[SWAP=0.8]novel/title</query>
  <query cd="0.3" book="" title="//">/bib/[DELETE=0.6][JUMP=0.5]//title</query>
  <query cd="0.2646" bib="//" bib="" book="" title="name">
                    /[JUMP=0.7]//[DELETE=0.7][DELETE=0.6][SWAP=0.9]name</query>
  .........
</result>
```

4 Implementation based on Fuzzy Logic Programming with MALP

Multi-Adjoint Logic Programming [MOV04], MALP in brief, is based on a first order language, $\mathscr{L}$, containing variables, function/constant symbols, predicate symbols, and several arbitrary connectives such as implications ($\leftarrow_1, \leftarrow_2, \ldots, \leftarrow_m$), conjunctions ($\&_1, \&_2, \ldots, \&_k$), disjunctions ($\vee_1, \vee_2, \ldots, \vee_l$), and general hybrid operators ("aggregators" $@_1, @_2, \ldots, @_n$), used for combining/propagating truth values through the rules, and thus increasing the language expressiveness. Additionally, our language $\mathscr{L}$ contains the values of a *multi-adjoint lattice* in the form $\langle L, \preceq, \leftarrow_1, \&_1, \ldots, \leftarrow_n, \&_n \rangle$, equipped with a collection of *adjoint pairs* $\langle \leftarrow_i, \&_i \rangle$ where each $\&_i$ is a conjunctor intended to the evaluation of *modus ponens*. A *rule* is a formula "$A \leftarrow_i \mathscr{B}$ with α", where A is an atomic formula (usually called the *head*), $\mathscr{B}$ (which is called the *body*) is a formula built from atomic formulas $B_1, \ldots, B_n$ ($n \geq 0$), truth values of L and conjunctions, disjunctions and general aggregations, and finally $\alpha \in L$ is the "weight" or *truth degree* of the rule. The set of truth values L may be the carrier of any complete bounded lattice, as for instance occurs with the set of real numbers in the interval $[0,1]$ with their corresponding ordering $\preceq_R$. Consider, for instance, the following program composed by three rules with associated multi-adjoint lattice $\langle [0,1], \preceq_R, \leftarrow_{\mathtt{P}}, \&_{\mathtt{P}} \rangle$ (where label P means for *Product logic* with the following connective definitions for implication, conjunction and disjunction symbols, respectively: "$\leftarrow_{\mathtt{P}}(x,y) = \min(1, x/y)$", "$\&_{\mathtt{P}}(x,y) = x * y$" and "$|_{\mathtt{P}}(x,y) = x + y - x * y$"):

$$\begin{array}{lllll} \mathscr{R}_1: & p(X) & \leftarrow_{\mathtt{P}} & q(X,Y) \mid_{\mathtt{P}} r(Y) & with \quad 0.8 \\ \mathscr{R}_2: & q(a,Y) & \leftarrow & & with \quad 0.9 \\ \mathscr{R}_3: & r(b) & \leftarrow & & with \quad 0.7 \end{array}$$

In order to run and manage MALP programs, during the last years we have designed the FLOPER system [MM08, MMPV10, MMPV11], which is freely accessible from the Web site http://dectau.uclm.es/floper/. The parser of our tool has been implemented by using the classical DCG's (*Definite Clause Grammars*) resource of the Prolog language, since it is a convenient notation for expressing grammar rules. Once the application is loaded inside a Prolog interpreter, it shows a menu which includes options for loading/compiling, parsing, listing and saving fuzzy programs, as well as for executing/debugging fuzzy goals. All these actions are based on the *compilation* of the fuzzy code into standard Prolog code.

The key point of the compilation is to extend each atom with an extra argument, called *truth variable* of the form "`_TV`$_i$", which is intended to contain the truth degree obtained after the subsequent evaluation of the atom. For instance, the first fuzzy rule of our previous program is translated to clause:

```
p(X,_TV0):-q(X,Y,_TV1),r(Y,_TV2),or_prod(_TV1,_TV2,_TV3),and_prod(0.8,_TV3,_TV0).
```

Moreover, the remaining rules become the pure Prolog facts "`q(a,Y,0.9)`" and "`r(b,0.7)`", whereas the corresponding lattice is expressed by the following Prolog clauses (where the meaning of the mandatory predicates `member`, `top`, `bot` and `leq` is obvious):

```
member(X):- number(X), 0=<X, X=<1.                          bot(0).
leq(X,Y):- X=<Y.                                           top(1).
and_prod(X,Y,Z):- pri_prod(X,Y,Z).    pri_prod(X,Y,Z):- Z is X * Y.
or_prod(X,Y,Z):- pri_prod(X,Y,U1),pri_add(X,Y,U2),pri_sub(U2,U1,Z).
pri_add(X,Y,Z):- Z is X+Y.               pri_sub(X,Y,Z):- Z is X-Y.
```

Finally, a fuzzy goal like "`p(X)`", is translated into the pure Prolog goal: "`p(X, Truth_degree)`" (note that the last truth degree variable is not anonymous now) for which, after choosing in FLOPER option "`run`", the underlying Prolog interpreter computes the desired fuzzy answer `[Truth_degree=0.776,X=a]`. Note that all internal computations (including compiling and executing) are pure Prolog derivations, whereas inputs (fuzzy programs and goals) and outputs (fuzzy computed answers) have always a fuzzy taste, thus producing the illusion on the final user of being working with a purely fuzzy logic programming tool. By using option "`lat`" ("`show`") of FLOPER, we can associate (and display) a new lattice to a given program. As seen before, such lattices must be expressed by means of a set of Prolog clauses (defining predicates `member`, `top`, `bot`, `leq` and the ones associated to fuzzy connectives) in order to be loaded into FLOPER.

4.1 MALP and XPath

MALP can be used as basis for our proposed fuzzy extension of XPath as follows. The idea is to implement XPath by means of MALP rules.

Firstly, we make use of a SWI-Prolog library for loading XML files in order to store each XML

document by means of a Prolog term[1] representing a tree. Each tag is represented as a data-term of the form: `element(Tag, Attributes, Subelements)`, where `Tag` is the name of the XML tag, `Attributes` is a list containing the attributes, and `Subelements` is a list containing the sub-elements (i.e. subtrees) of the tag. For instance, the XML document of Figure 3 is represented in SWI-Prolog as:

```
[element(bib, [],
   [element(name,[],['Classic Literature']),
    element(book, [year='2001',price='45.95'],
      [element(title,[],['Don Quijote de la Mancha']),
       element(author,[],['Miguel de Cervantes Saavedra']),
       element(references,[],
         [element(novel,[year='1997',price='35.99'],
            [element(name,[],['La Galatea']),
             element(author,[],['Miguel de Cervantes Saavedra']),
                 element(references,[],
           ...])
   ...]),
])])
```

Secondly, for loading XML documents in our implementation we make use of the SWI-Prolog predicate `load_xml(+File,-Term)`. Similarly, we have a predicate `write_xml(+File,+Term)` for writing a data-term representing an XML document into a file.

Thirdly, we have to consider a complete lattice L_{tv} to be used for representing the RSV's and the CD's. L_{tv} contains trees, which we will call *tv trees*, of the form: "$tv(v,[root,tvch,tvsib])$", where v is a value taken from $[0,1]$ (i.e., the RSV or CD), *root* is the root of the tree to which v is associated, and *tvch*, *tvsib* are the *tv trees* of the children and sibling nodes, respectively.

Finally, MALP rules have the form "$A \leftarrow \mathscr{B}$ *with tvtree*". In addition, some XPath fuzzy operators *and*, *or* and *avg* can be mapped to MALP connectives in lattice L_{tv}. Finally, we have to consider an auxiliary aggregator in L_{tv} called @*fuse*, which builds the *tv tree* of a certain node from the tv trees of the sibling and children nodes. For instance, some representative Prolog clauses (used in FLOPER) defining the new lattice based on *tv trees* are:

```
member(tv(N,L)):-number(N),0=<N,N=<1,(L=[];L=[_|_]).          bot(tv(0,[])).
and_prod(tv(X1,X2),tv(Y1,Y2),tv(Z1,Z2)):-pri_prod(X1,Y1,Z1),pri_app(X2,Y2,Z2).
pri_app([],X,X).                    pri_app([A|B],C,[A|D]):-pri_app(B,C,D).
```

4.2 MALP and the XPath Debugger

The core of our debugger is coded with MALP rules by reusing most modules of our fuzzy XPath interpreter [ALM11, ALM12]. And, of course, the parser of our debugger has been extended to recognize the new keywords `DEBUG`, `SWAP`, `DELETE` and `JUMP`, with their proper arguments.

Now, we would like to show how the new "*XPath debugging*" predicate admits an elegant definition by means of fuzzy MALP rules. Each rule defining predicate:

$$\texttt{debugQuery}(\texttt{ListXPath},\texttt{Tree},\texttt{Penalty})$$

receives three arguments: (1) `ListXPath` is the Prolog representation of an XPath expression, (2) `Tree` is the term representing an input XML document and (3) `Penalty` represents the *chance degree*. A call to this predicate returns a truth-value (i.e., a *tv tree*) like the following one:

[1] The notion of *term* (i.e., data structure) is just the same in MALP and Prolog.

```
tv(1.0,[[/,[]],
   tv(1.0, [[tag(bib),[]],
      tv(1.0,[[tag(book),[]],
         tv(1.0,[[tag(title),[]],[],
         ...
      tv(0.8,[[tag(novel),[book=novel]],
      ...
      tv(0.5,[['[DELETE=0.5]',[book='']],
         tv(0.9,[[tag(name),[title=name]],[],
         ...
         []])]),
      [])]),
   [])])])]),
[])
```

Basically, the *debugQuery* predicate traverses the XML document, checking the validity of the original query *tag-by-tag*, and trying to match each *tag* to the ones occurring in the XML document and thus producing effects of DELETE, JUMP and SWAP.

The definition of such predicate includes several rules for distinguishing the three debugging cases. As an example the case of swapping is defined as follows:

```
debugQuery([Label|LabelRest],[element(Label2,_,Children)|Siblings],Penalty) <prod
       @fuse(
               similarity(Label, Label2),
               debugQuery(LabelRest, Children, Penalty),
               debugQuery([Label|LabelRest], Siblings, Penalty)
       ) with tv(1,[]).
```

which becomes into the following Prolog clause after compilation into FLOPER:

```
debugQuery([Label|LabelRest],[element(Label2,_,Children)|Siblings], Penalty, TV_Iam):-
       similarity(Label, Label2, TV_Similarity),
       debugQuery(LabelRest, Children, Penalty, TV_Son),
       debugQuery([Label|LabelRest], Siblings, Penalty, TV_Sib),
       agr_fuse(TV_Similarity, TV_Son, TV_Sib, TV_Iam).
```

Basically, similarity is checked with the atom `similarity(Label, Label2)`, and two recursive calls are achieved for debugging both children (`debugQuery(LabelRest, Children, Penalty)`) and siblings (`debugQuery([Label|LabelRest], Siblings, Penalty, 1)`), whose *tv trees* are finally combined (*fused*) with the node content.

Finally, for showing the result in a pretty way (see Figures 7), and transforming a *tv tree* into an XML file, a predicate *tv_to_element* has been implemented.

5 Conclusions and Future Work

In this paper we have presented a proposal of XPath debugging. The result of the debugging process of a XPath expression is a set of alternative queries, each one associated to a chance degree. We have proposed JUMP, DELETE and SWAP operators that cover the main cases of programming errors when describing a path about an XML document. We have implemented and tested the approach. The approach has a fuzzy taste in the sense that XPath expressions are debugged by relaxing the path expression assigning a chance degree to debugging points. The fuzzy taste is also illustrated when executing the annotated XPath expressions in a fuzzy based implementation of XPath.

Although XML and XPath are extensively used in many applications, the debugging of XPath has not been explored enough in the literature. Some authors have explored this topic [ACGS12], where the functional logic language TOY has been used for debugging XPath expressions. There the debugger is able to assist the programmer when a tag is wrong, providing alternative tags, and to trace executions. The current work can be considered as an extension of the quoted work. Here, the debugging gives to programmers a chance degree for each tag alternative, and annotates XPath expressions in the points in which the error was found. We have based our work in the proposal on [FFF10, FFF11] where XPath relaxation is studied given some rules for query rewriting: axis relaxation, step deletion and step cloning, among others. However, they do not give chance degree associated to the input XPath expression.

Our future work will focus on incorporating new elements to the debugging. The first one is the handling of boolean conditions. Boolean conditions can express ranges that can be wrong and conditions that cannot be satisfied. The second one is to provide more flexibility to the debugger language: `DEBUG` is used in the debugger to annotate the penalization, and it is used for `JUMP`, `DELETE` and `SWAP`. A more flexible case would be to provide a different penalization in each case.

Acknowledgements: This work has been partially supported by the EU, under FEDER, and the Spanish Science and Innovation Ministry (MICINN) under grants TIN2008-06622-C03-03 and TIN2007-65749, as well as by Ingenieros Alborada IDI under grant TRA2009-0309, the Castilla-La Mancha Administration under grant PII1I09-0117-4481, and the JUNTA ANDALUCIA administration under grant TIC-6114 (proyecto de excelencia).

Bibliography

[ACGS12] J. M. Almendros-Jiménez, R. Caballero, Y. GarcíA-Ruiz, F. SáEnz-PéRez. XPath Query Processing in a Functional-Logic Language. *Electron. Notes Theor. Comput. Sci.* 282:19–34, May 2012.

[ALM11] J. Almendros-Jiménez, A. Luna, G. Moreno. A Flexible XPath-based Query Language Implemented with Fuzzy Logic Programming. In *Proc. of 5th International Symposium on Rules: Research Based, Industry Focused, RuleML'11. Barcelona, Spain, July 19–21*. Pp. 186–193. Springer Verlag, LNCS 6826, Heidelberg,Germany, 2011.

[ALM12] J. M. Almendros-Jiménez, A. Luna, G. Moreno. Fuzzy Logic Programming for Implementing a Flexible XPath-based Query Language. *Electr. Notes Theor. Comput. Sci.* 282:3–18, 2012.

[BBC+07] A. Berglund, S. Boag, D. Chamberlin, M. Fernandez, M. Kay, J. Robie, J. Siméon. XML path language (XPath) 2.0. *W3C*, 2007.

[FFF10] B. Fazzinga, S. Flesca, F. Furfaro. On the expressiveness of generalization rules for XPath query relaxation. In *Proceedings of the Fourteenth International Database Engineering & Applications Symposium*. Pp. 157–168. 2010.

[FFF11] B. Fazzinga, S. Flesca, F. Furfaro. XPath Query Relaxation through Rewriting Rules. *IEEE transactions on knowledge and data engineering* 23(10):1583–1600, 2011.

[MM08] P. Morcillo, G. Moreno. Programming with Fuzzy Logic Rules by using the FLOPER Tool. In al. (ed.), *Proc of the 2nd. Rule Representation, Interchange and Reasoning on the Web, International Symposium, RuleML'08*. Pp. 119–126. Springer Verlag, LNCS 3521, 2008.

[MMPV10] P. Morcillo, G. Moreno, J. Penabad, C. Vázquez. A Practical Management of Fuzzy Truth Degrees using FLOPER. In al. (ed.), *Proc. of 4nd Intl Symposium on Rule Interchange and Applications, RuleML'10*. Pp. 20–34. Springer Verlag, LNCS 6403, 2010.

[MMPV11] P. Morcillo, G. Moreno, J. Penabad, C. Vázquez. Fuzzy Computed Answers Collecting Proof Information. In al. (ed.), *Advances in Computational Intelligence - Proc of the 11th International Work-Conference on Artificial Neural Networks, IWANN 2011*. Pp. 445–452. Springer Verlag, LNCS 6692, 2011.

[MOV04] J. Medina, M. Ojeda-Aciego, P. Vojtáš. Similarity-based Unification: a multi-adjoint approach. *Fuzzy Sets and Systems* 146:43–62, 2004.

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): Actas de las "XII Jornadas sobre Programación y Lenguajes (PROLE'2012), Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

Sesión 4

Programación lógica/restricciones

Chair: *Dr. Fernando Orejas*

Sesión 4: Programación lógica/restricciones
Chair: Dr. Fernando Orejas

Pablo Chico De Guzmán, Manuel Carro, Manuel Hermenegildo and Peter Stuckey. *A General Implementation Framework for Tabled CLP*.

Marco Comini, Laura Titolo and Alicia Villanueva. *Abstract diagnosis for timed concurrent constraint programs*.

Miquel Bofill Arasa, Joan Espasa Arxer, Miquel Palahí Sitges and Mateu Villaret. *An extension to Simply for solving Weighted Constraint Satisfaction Problems with Pseudo-Boolean Constraints*.

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): Actas de las "XII Jornadas sobre Programación y Lenguajes (PROLE'2012), Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

A General Implementation Framework for Tabled CLP

Pablo Chico de Guzmán[1] **Manuel Carro**[1,2]
Manuel V. Hermenegildo[1,2] **Peter Stuckey**[3,4]

[1]IMDEA Software Institute, Spain [2]School of Computer Science, UPM, Spain
[3]NICTA Laboratory, Australia [4]School of Computer Science, U. of Melbourne, Australia

Abstract: This is a summary of [PCHS12], where a framework to combine tabling evaluation [TS86, War92] and constraint logic programming [JM94] is described (TCLP). This combination has been studied previously from a theoretical point of view [Tom97, Cod95], where it is shown that the constraint domain needs to offer projection and entailment checking operations in order to ensure completeness w.r.t. the declarative semantics. However, existing TCLP frameworks and implementations lack a complete treatment of constraint projection and / or entailment. The novelty of our proposal is that *we present a complete implementation framework for TCLP, independent from the constraint solver, which can use either precise or approximate projection and entailment, possibly with optimizations.*

Keywords: Constraint Logic Programming, Tabling, Implementation, Performance.

Tabled evaluation is an execution strategy for logic programs that records calls and their answers in order to reuse them in future calls. The operational semantics of tabled LP differentiates the first call to a tabled predicate, a *generator*, from subsequent identical calls, the *consumers*. Generators resolve against program clauses and insert the answers they compute in a global table. Consumers read answers from the global table and suspend when no more answers are available (therefore breaking infinite loops) and wait for the generation of more answers by their generators. A generator is said to be *complete* when it is known not to be able to generate more (unseen) answers. In order to check this property, a fixpoint procedure is executed where all the consumers inside the generator execution subtree are reading their pending answers until no more answers are generated. Due to the fixpoint computation of the completion operation, tabled LP is useful in several scenarios. The following program computes lengths of paths in a graph:

```
:− table path/3.
path(X,Y, 1) :− edge(X,Y).
path(X,Y, L) :− edge(X,Z), path(Z,Y,L2), L is L2 + 1.
edge(a,a).
```

where edge/2 defines the graph connectivity, the query ?− path(a,*Y*,*L*). returns the length of the paths from node a to all its reachable nodes, even if the graph has cycles. The previous program can be rewritten using constraints in order to return only those paths whose length is less than a given number. To this end, the third line of the program turns into the line:

```
path(X,Y, L) :− edge(X,Z), L #= L2 + 1, path(Z,Y,L2).
```

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): PROLE'2012, pp. 137-138, ISBN:978-84-15487-27-2.
Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

Now, the query ?− L#=< 10, path(a,Y,L). creates a generator whose first clause computes the answer $\{Y$=a, L=1$\}$. On backtracking, the second clause of path/3 calls path(a,Y,$L2$), where $L2$ is knows to be less or equal than 9. This call is more specific than the previous generator and it can be considered as a consumer; this is detected using entailment. A set of constraints C_1 is entailed by another set of constraints C_2 in the domain D if $D \models C_2 \rightarrow C_1$. In order to check for entailment, the constraints associated with variables in the store but not in the call (e.g., L) should be ignored. This is done via projection on the variables of the tabled call. The projection of constraint C onto variables V is a constraint C' over variables V such that $D \models \exists \bar{x}.C \leftrightarrow C'$ where $\bar{x} = vars(C) - V$. The projection and entailment operations ensure soundness and termination for "constraint-compact" domains of constraints under tabled evaluation.

Our tabled CLP framework builds on a usual tabling system, providing checking of Herbrand terms (to avoid repeated tabled calls/answers) and consumer suspension/resumption, and defines an interface to be implemented by the constraint solver. The operations of this interface are responsible for managing the different constraint sets for tabled calls/answers with identical Herbrand terms (up to variable renaming). Answer merging can be also implemented to optimize the memory consumption of answer memoing. Therefore, our tabled CLP is independent from the constraint solver. We have validated the flexibility and generality of our framework (with excellent performance results) by implementing two examples: difference constraints and disequality constraints. Among others, tabled CLP can be applied to constraint databases, model checking of timed automata and abstract interpretation.

Acknowledgements: Work partially funded by EU projects IST-215483 *S-Cube* and FET IST-231620 *HATS*, MICINN projects TIN-2008-05624 *DOVES*, and CAM project S2009TIC-1465 *PROMETIDOS*. Pablo Chico was also funded by a MICINN FPU scholarship.

Bibliography

[Cod95] P. Codognet. A Tabulation Method for Constraint Logic Programming. In *INAP'95*. Tokyo, Japan, Oct. 1995.

[JM94] J. Jaffar, M. Maher. Constraint LP: A Survey. *JLP* 19/20:503–581, 1994.

[PCHS12] P. Chico de Guzmán, M. Carro, M. Hermenegildo, P. Stuckey. A General Implementation Framework for Tabled CLP. In Schrijvers and Thiemann (eds.), *FLOPS'12*. LNCS 7294, pp. 104–119. Springer Verlag, May 2012.

[Tom97] D. Toman. Memoing Evaluation for Constraint Extensions of Datalog. *Constraints* 2(3/4):337–359, 1997.

[TS86] H. Tamaki, M. Sato. OLD Resol. with Tabulation. In *ICLP*. Pp. 84–98. LNCS, 1986.

[War92] D. S. Warren. Memoing for Logic Programs. *CACM* 35(3):93–111, 1992.

Abstract Diagnosis for Timed Concurrent Constraint programs - Abstract

M. Comini[1] **L. Titolo**[1] **A. Villanueva**[2*]

Dipartimento di Matematica e Informatica, U. di Udine[1]
DSIC, Universitat Politècnica de València[2]

Abstract:

This short paper is a summary of the published paper [CTV11] where a general framework for the debugging of *tccp* programs is defined. To this end, a new compact, bottom-up semantics for the language that is well suited for debugging and verification purposes in the context of reactive systems was presented. In order to effectively implement the technique, we also provided an abstract semantics.

Keywords: concurrent constraint paradigm, denotational semantics, abstract diagnosis, abstract interpretation

1 Abstract Diagnosis for *tccp*

Finding program bugs is a long-standing problem in software construction. In the concurrent paradigms, the problem is even worse and the traditional tracing techniques are almost useless. There has been a lot of work on algorithmic debugging for declarative languages, which could be a valid proposal for concurrent paradigms, but little effort has been done for the particular case of the concurrent constraint paradigm (*ccp* in short; [Sar93]). The *ccp* paradigm is different from other programming paradigms mainly due to the notion of store-as-constraint that replaces the classical store-as-valuation model. In this way, the languages from this paradigm can easily deal with partial information: an underlying constraint system handles constraints on system variables. Within this family, [BGM00] introduced the *Timed Concurrent Constraint Language* (*tccp* in short) by adding to the original *ccp* model the notion of time and the ability to capture the absence of information. With these features, it is possible to specify behaviors typical of reactive systems such as *timeouts* or *preemption* actions, but they also make the language non-monotonic.

We develop an abstract diagnosis method for *tccp* using the ideas of the abstract diagnosis framework for logic programming [CLMV99]. This framework, parametric w.r.t. an abstract program property, is based on the use of an abstract immediate consequence operator to identify bugs in logic programs. The intuition of the approach is that, given an abstract specification of the expected behavior of the program, one automatically detects the errors in the program. The framework does not require the determination of symptoms in advance. In order to achieve an effective method, abstract interpretation is used to approximate the semantics, thus results may be less precise than those obtained by using the concrete semantics.

* This work has been partially supported by the EU (FEDER), the Spanish MICINN under grant TIN2010-21062-C02-02 and by Generalitat Valenciana, ref. PROMETEO2011/052.

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): PROLE'2012, pp. 139-140, ISBN:978-84-15487-27-2.
Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

Previous work in the literature, has shown that a key point for the efficacy of the resulting debugging methodology is the compactness of the concrete semantics. Thus, we have devoted much effort to the development of a compact concrete semantics for the *tccp* language to start with. There are many languages where a compact compositional semantics has been founded on collecting the possible traces for the weakest store, since all traces relative to any other initial store can be derived by instance of the formers. In *tccp*, this does not work since the language is not monotonic: if we have all traces for an agent A starting from an initial store c and we execute A with a more instantiated initial store d, then new traces, not instances of the formers, can appear, and others (that were *compatible* with c but not with d) may disappear.

Furthermore, note that, since we are interested in a bottom-up approach, we cannot work assuming that we know the initial store. However, when we define the semantics of a conditional or choice agent where some guard must be checked, we should consider different execution branches depending on the guard satisfiability. To deal with all these particular features, our idea is that of associating conditions to computation steps, and to collect all possible minimal hypothetical computations.

Our new (concrete) compact *compositional* semantics is correct and fully abstract w.r.t. the small-step behavior of *tccp*. It is based on the evaluation of agents over a denotation for a set of process declarations D, obtained as least fixpoint of a (continuous, monotone) immediate consequence operator $\mathscr{D}[\![D]\!]$.

Thanks to the compactness of this semantics, we can formulate an efficacious debugging methodology based on abstract interpretation which proceeds by approximating the $\mathscr{D}[\![D]\!]$ operator producing an "abstract immediate consequence operator" $\mathscr{D}^{\alpha}[\![D]\!]$. We show that, given the abstract intended specification $\mathscr{S}^{\alpha}$ of the semantics of the declarations D, we can check the correctness of D by a single application of $\mathscr{D}^{\alpha}[\![D]\!]$ and thus, by a static test, we can determine all the process declarations $d \in D$ which are wrong w.r.t. the considered abstract property.

When applying the diagnosis w.r.t. approximate properties, the results may be weaker than those that can be achieved on concrete domains just because of approximation. Abstract incorrect process declarations are in general just a warning about a possible source of errors. Because of the approximation, it can happen that a (concretely) correct declaration is abstractly incorrect. However, all concrete errors are detected.

Bibliography

[BGM00] F. S. de Boer, M. Gabbrielli, M. C. Meo. A Timed Concurrent Constraint Language. *Information and Computation* 161(1):45–83, 2000.

[CLMV99] M. Comini, G. Levi, M. C. Meo, G. Vitiello. Abstract Diagnosis. *Journal of Logic Programming* 39(1-3):43–93, 1999.

[CTV11] M. Comini, L. Titolo, A. Villanueva. Abstract Diagnosis for Timed Concurrent Constraint programs. *Theory and Practice of Logic Programming* 11(4-5):487–502, 2011 2011.

[Sar93] V. A. Saraswat. *Concurrent Constraint Programming*. The MIT Press, Cambridge, Mass., 1993.

An extension to Simply for solving Weighted Constraint Satisfaction Problems with Pseudo-Boolean Constraints

Miquel Bofill, Joan Espasa, Miquel Palahí, and Mateu Villaret

Departament d'Informàtica i Matemàtica Aplicada
Universitat de Girona, Spain
{mbofill,jespasa,mpalahi,villaret}@ima.udg.edu

Abstract: Max-Simply is a high-level programming framework for modelling and solving weighted CSP. Max-Simply can also deal with meta-constraints, that is, constraints on constraints. The technology currently used to solve the generated problem instances is SMT. In this paper we present a variant of Max-Simply which is able to generate not only SMT instances but also pseudo-Boolean instances for certain modellings. Since there are problems that are more naturally encoded using pseudo-Boolean variables, the possibility of generating pseudo-Boolean instances can result in a more efficient and natural fit in some situations. We illustrate the expressiveness of the Max-Simply language by modelling some problems, and provide promising performance results on the corresponding generated pseudo-Boolean instances using state-of-the-art pseudo-Boolean solvers.

Keywords: Modelling languages, Pseudo-Boolean constraints, CSP, Weighted CSP.

1 Introduction

One of the challenges in constraint programming is to develop systems allowing the user to easily specify the problem in a high-level language and, at the same time, being able to efficiently solve it. Various approaches exist for solving constraint satisfaction problems (CSP) specified in a high-level language, such as ad hoc algorithms for some constructs, or the translation to lower level languages. Some years ago, translation was seen as a theoretical possibility but not really feasible. But there have been impressive developments in this area, making this approach not only feasible, but also competitive.

Following this direction, some high-level, solver-independent constraint modelling languages have been developed, such as MiniZinc [NSB+07], ESSENCE [FHJ+08] and `Simply` [BPSV09]. Those languages let the user express most CSPs easily and intuitively. There exist tools for translating from those high-level languages to lower level ones, some of them permitting a great deal of flexibility to the user. For example, the MiniZinc system lets the user specify which constraints wants to leave untranslated, so that an underlying solver or an ad hoc algorithm can deal with them in a better and more efficient way.

`Simply` was developed as a declarative programming system for easy modelling and solving of CSPs. Essentially, the system translates CSP instances (written in its own language) into satisfiability modulo theories (SMT) instances, which are then fed into an SMT solver. SMT instances generalize Boolean formulas by allowing the use of predicates with predefined interpretations from background theories such as, e.g., linear integer arithmetic. For example, a

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): PROLE'2012, pp. 141-155, ISBN:978-84-15487-27-2.
Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

formula can contain clauses like $p \vee q \vee (x+2 \leqslant y) \vee (x > y+z)$, where p and q are Boolean variables and x, y and z are integer variables. `Max-Simply` [ABP+11] is an extension to `Simply`, allowing to model and solve weighted CSPs.

There exist many problems that are easily or naturally encoded using pseudo-Boolean variables, i.e., integers with a $\{0,1\}$ domain. For those kind of problems, very efficient pseudo-Boolean solvers exist. Here we propose a variant of `Max-Simply`, where we add the possibility to translate most of its high-level language constructs into low-level pseudo-Boolean constructs, allowing the user to solve his modelled weighted CSP problems using an even wider spectrum of solvers and technologies. This extension also provides support for some constraints on constraints, so called meta-constraints.

The rest of the paper is structured as follows. In Section 2, weighted CSP and pseudo-Boolean constraints are presented, including the pseudo-Boolean optimization problem and the weighted Boolean optimization problem. In Section 3, the language of `Simply` is briefly described, empathizing its features and how the translation to pseudo-Boolean constraints works. In Section 4, results of the experiments with two different problems are shown and analyzed. In Section 5, conclusions are drawn from the results so far and some future work is proposed.

2 Preliminaries

2.1 Weighted CSP

A *constraint satisfaction problem (CSP)* instance is defined as a triple $\langle X, D, C \rangle$, where X is a set of variables, D is a set of domains containing the values the variables may take, and C is a set of constraints. Each constraint is defined as a relation over a subset of variables. A constraint or relation may be represented *intensionally*: in terms of an expression that defines the relationship that must hold amongst the assignments to the variables it constrains, or it may be represented *extensionally*: as explicitly enumerating the allowed assignments or the disallowed assignments. An *assignment* for a CSP instance $\langle X, D, C \rangle$ is a mapping that assigns to every variable in X an element that belongs to its domain, and a *solution* is an assignment that satisfies the constraints.

A *weighted CSP (WCSP)* is defined as a CSP, where each constraint has attached a *weight*, or falsification *cost* (typically a positive number), expressing the penalty for violating the constraint. An optimal solution to a WCSP instance is a complete assignment in which the sum of the weights of the violated constraints is minimal. We call *hard* those constraints whose associated cost is infinity, *soft* otherwise. For further formal definitions the reader can refer to [LS04].

2.2 Pseudo-Boolean constraints

A pseudo-Boolean constraint (PBC for short) is an inequality on a linear combination of 0-1 (Boolean) variables. PBCs take the form:

$$a_1 x_1 + a_2 x_2 + \cdots + a_n x_n \oplus r \tag{1}$$

where $a_1, \ldots, a_n$ and r are integer constants, $x_1, \ldots, x_n$ are Boolean variables, and $\oplus$ is a relational operator in $\{<, >, \leqslant, \geqslant, =\}$.

So a system of PBCs is a system of linear inequalities of 0-1 variables, a paradigm extensively studied in Operations Research. PBCs lie on the border between satisfiability testing, constraint programming, and integer programming. With their expressivity, PBCs can be used to compactly describe many discrete electronic design automation problems, network layout problems, and much more.

The yearly pseudo-Boolean competition [RM12] works as a place to compare the efficiency of the current pseudo-Boolean solvers. The competition is divided in two branches, one for pseudo-Boolean optimization and one for weighted Boolean optimization. Each branch has its own file format. For solving pseudo-Boolean optimization problems, there are three big families of approaches: integer programming, pseudo-Boolean resolution and translation into MaxSAT. For further information, the reader can refer to [RM09].

2.2.1 Pseudo-Boolean optimization

Pseudo-Boolean optimization (PBO) is a natural extension of the Boolean satisfiability problem (SAT). A PBO instance is a set of PBC as defined in Equation 1, plus a linear objective function taking a similar form:

$$min: \ a_1x_1 + a_2x_2 + ... + a_nx_n \tag{2}$$

where $a_1,...,a_n$ are integer coefficients and $x_1,...,x_n$ are Boolean variables. The objective is to find a suitable assignment to the problem variables such that all the constraints are satisfied and the value of the objective function is minimized.

The first approaches into solving PBO instances were the generalization of the most effective techniques used in SAT solving, such as conflict-based learning, and conflict-directed backtracking [MM02, CK05]. The basic idea is, when a solution is found, to add a constraint such as only a lower value solution can be accepted. The solver finishes when a lower value solution cannot be found, thus proving its optimality.

Another common approach is the use a branch and bound algorithm, where the lower bounding procedure tries to estimate the value of the objective function. Direct translations from PBC to SAT have also been used effectively, especially in problem sets where the translation does not grow much larger than the original encoding.

The standard file format used in the pseudo-Boolean competition [RM12] is `opb`. This is a simple format, which is accepted by all the solvers. As a reference, a trivial example of this format can be seen in Figure 1. The first line of the file defines the number of variables and constraints in the problem instance.

```
* #variable= 2 #constraint= 1
**************************************
min: +1 x1 +1 x2;
+1 x1 +6 x2 >= -1;
```

Figure 1: `opb` example.

2.2.2 Weighted Boolean optimization

Weighted Boolean optimization (WBO) is another way of expressing a pseudo-Boolean optimization problem. This format aggregates and extends PBO and MaxSAT. It is composed of two types of constraints: hard and soft. Each soft constraint has an integer associated to it, representing its falsification cost. Given a WBO instance, an optimal solution is to find an assignment to the variables that satisfies all the hard constraints and minimizes the total cost of the unsatisfied soft constraints. Generalizations of MaxSAT and PBO algorithms are the first approaches that have been proposed to solve this kind of problems [MMP09].

The standard file format used in the pseudo-Boolean competition [RM12] for WBO instances is `wbo`. This is a simple format, accepted by most of the solvers. As a reference, a trivial example of this format can be seen in Figure 2. The first line of the file defines the number of variables and constraints, the number of soft constraints, the minimum and maximum cost of a single constraint and the total sum. The second line is defined as the sum of the costs of all soft constraints plus one.

```
* #variable= 3 #constraint= 4 #soft= 2 mincost= 2 maxcost= 3 sumcost= 5
soft: 6
+1 x1 +1 x2 +1 x3 >= 2;
+2 x1 +1 x2 +1 x3 >= 2;
[2] +1 x1 +1 x2 >= 1;
[3] +1 x1 +1 x3 >= 1;
```

Figure 2: `wbo` example.

3 Simply: extensions and variants

`Simply`[1] [BPSV09] is a programming system with a declarative language (also called `Simply`) for easy modelling and solving of CSPs, which uses an SMT solver as its core solving engine. Essentially, the system consists of a compiler from the `Simply` language into SMT, and a recovery module for translating the output generated by the SMT solver to the original problem format. Currently, `Simply` is integrated with the state-of-the-art SMT solver Yices [DM06], using its built-in API. However, it can be easily adapted to work with other SMT solvers, either using external files or any other API. Since most SMT solvers can only deal with decision problems, in the case of optimization `Simply` repeatedly calls the decision procedure performing binary search on the value of the objective function (treated as a constraint).

Recently, `Simply` has been extended to deal with weighted CSPs as well as with meta-constraints, in a new system called `Max-Simply` [ABP+11]. In this extension, the user can choose from the command line whether to solve the weighted CSP instance at hand (i) through successive calls to the decision procedure doing binary search on the corresponding objective function, or (ii) reformulating it into a weighted SMT instance and directly solving it with Yices (which has built-in support for weighted SMT), or (iii) applying the WPM1 algorithm [ABL09] on the generated weighted SMT instance.

[1] The tool is available from http://ima.udg.edu/recerca/LAP/simply/

In this paper we present a variant of `Max-Simply` which uses a pseudo-Boolean solver (PB solver for short) as its core solving engine. Although the pseudo-Boolean language is less expressive than that of SMT, solving CSP instances with a PB solver greatly simplifies the solving approach for optimization problems, due to the fact that PB solvers directly support optimization of objective functions. In the case of dealing with weighted CSPs, the architecture of the system can remain exactly the same as in `Max-Simply`, because most PB solvers support both optimization functions and weighted expressions. Consequently, the weighted instances can be directly reformulated either into PBO or WBO.

3.1 The language

The language of `Simply` is similar to other high-level modelling languages such as those of ESSENCE and MiniZinc. As it is usual, the model and the data of the problem instance can be stored in two different files. The decision variables in `Simply` can be finite domain integer variables and Boolean variables. For the pseudo-Boolean translation, only integer variables with the binary domain $\{0,1\}$ are currently considered.

Problems can be modelled by posting basic constraints, and also global and meta-constraints, that are decomposed into basic ones during the translation. The posting can be made either directly or by using generators such as `Forall` and `Exists`, and the `If-Then-Else` filter. The language also has the declarative facility of list comprehensions, allowing the user to generate parametrized lists of elements or constraints.

The basic constraints consist of arithmetic relational expressions built up with the usual operators $\leq$, $\geq$, $<$, $>$ and $=$ (only linear expressions are supported). These can be combined with other Boolean expressions using the Boolean operators `Or`, `And`, `Xor`, `Iff` and `Implies`, as for example in `b Implies 3*a[1]+4*a[2]+5*a[3] <= 8`.

The most representative global constraints supported by `Simply` are the following:

- `Sum(`l`,`n`)`. The sum of the elements in list l is n.
- `AtLeast(`l`,`v`,`n`)`. The number of occurrences of value v in list l is at least n.
- `AtMost(`l`,`v`,`n`)`. The number of occurrences of value v in list l is at most n.

In all the constraints, the elements of the lists, as well as the values v and n, must be linear integer expressions. In the present variant with pseudo-Boolean variables, the elements of the lists are restricted to expressions of the form $a*x$, where a is an integer constant and x is a pseudo-Boolean variable. Moreover, the last two global constraints have been modified in the following way (with the second argument removed):

- `AtLeast(`$[a_1*x_1,\ldots,a_n*x_n]$`,`n`)` if and only if $\sum_{i=1}^{n} a_i*x_i \geq n$.
- `AtMost(`$[a_1*x_1,\ldots,a_n*x_n]$`,`n`)` if and only if $\sum_{i=1}^{n} a_i*x_i \leq n$.

Example 1 *If we want to constraint the sum of the elements in a list of three elements to be equal to* 10*, we can post the basic constraint* `list[1] + list[2] + list[3] = 10;` *However, if we don't know, a priori, the length of the list because it is an integer constant value* `n` *of the instance data, we can post the following global constraint, where we use a list comprehension to create a list of unknown length:* `Sum([ list[i] | i in [1..n] ], 10);`

The generator

$$\texttt{Forall}(id\ \texttt{in}\ list)\ \{constraints\}$$

posts a conjunction of the generated inner constraints (either basic or global), while the generator

$$\texttt{Exists}(id\ \texttt{in}\ list)\ \{constraints\}$$

posts a disjunction. The language also has the filter

$$\texttt{If}(formula)\ \texttt{Then}\ \{constraints_1\}\ \texttt{Else}\ \{constraints_2\};$$

where the condition *formula* is evaluated in compilation time. If it evaluates to true then the $constraints_1$ are posted; otherwise the $constraints_2$ are posted.

Example 2 *The following code with two generators and one filter*

```
% Every element of the list must be between 0 and 10
Forall(i in [1..n]) {
  list[i] >= 0; list[i] =< 10;
};
% One of the elements ocurring in even positions
% of the list must be greater than or equal to 7
Exists(i in [1..n]) {
  If (n Mod 2 == 0) Then {
    list[i] >= 7;
  };
};
```

results into the following constraints, provided that the value of `n` *is known to be* `5` *at compile time:*

```
list[1] >= 0; list[1] =< 10;
list[2] >= 0; list[2] =< 10;
list[3] >= 0; list[3] =< 10;
list[4] >= 0; list[4] =< 10;
list[5] >= 0; list[5] =< 10;
(list[2] >= 7 Or list[4] >= 7);
```

The language also allows the use of list comprehensions as a `Forall` generator, i.e., list comprehensions can be used to post basic constraints.

Example 3 shows a `Simply` model of a toy instance of the Nurse scheduling problem (NSP).

Example 3 *Consider a simple instance of the NSP (details on the NSP are given in Section 4.3) with two shifts per day and two available nurses. Each shift must be covered with exactly one nurse. We also want to satisfy the preferences of the nurses. Say nurses 1 and 2 want to work both days on shift 1. This particular instance can be modelled with* `Simply` *as follows:*

```
Problem:nsp
   Data
         int nurses = 2; int days = 2; int shifts = 2;
   Domains
         Dom pseudo = [0..1];
   Variables
         IntVar nd[nurses, days, shifts] :: pseudo;
   Constraints
         % Exactly one nurse per day and shift
```

```
Forall(d in [1..days], st in [1..shifts]) {
   Sum([nd[n, d, st] | n in [1..nurses]], 1); };

% Exactly one shift per nurse and day
Forall(d in [1..days], n in [1..nurses]) {
   Sum([nd[n, d, st] | st in [1..shifts]], 1); };

% Preferences for nurse 1
nd[1,1,1] = 1; % work day 1 shift 1
nd[1,2,1] = 1; % work day 2 shift 1

% Preferences for nurse 2
nd[2,1,1] = 1; % work day 1 shift 1
nd[2,2,1] = 1; % work day 2 shift 1
```

We can identify four sections in the `Simply` *model: data, domains, variables and constraints. The data section allows us to define a particular instance of the problem. In the variables section we declare the decision variables of the problem, in this case a tridimensional array of pseudo-Boolean variables* `nd`*, where the size of the first dimension corresponds to the number of nurses, the size of the second dimension to the number of days and the size of the third dimension to the number of shifts. This array states whether each nurse works in each shift of each day. In the constraints section, we first introduce the cover constraints: for every day, we have to ensure that every shift is covered by the required number of nurses (one in this case). To post this constraint we generate the list of nurses working each day* `d` *and shift* `st` *with a list comprehension* `[nd[n, d, st] | n in [1..nurses]]`*, and we use the global constraint* `Sum` *to require the number of nurses working that day and shift to be exactly one. Similarly we post a second constraint for the number of shifts worked by a nurse in one day. Finally, we add the preference constraints of each nurse for each day, specifying which shift they prefer. As we can see this toy instance is unsatisfiable, since all nurses want to work the same shift.*

3.1.1 Soft constraints and meta-constraints

As we have explained at the beginning of this section, `Max-Simply` can deal with weighted CSPs, so the language implements *soft* constraints. For instance, in order to relax the nurse preference constraints of the previous example we only need to turn those hard constraints into soft constraints, by adding their weights as follows:

```
#A: nd[1,1,1] = 1 @{1};  #B: nd[1,2,1] = 1 @{1};
#C: nd[2,1,1] = 1 @{1};  #D: nd[2,2,1] = 1 @{1};
```

where the `@{1}` are the corresponding weights, and where `A`, `B`, `C` and `D` are labels that may be used later on to refer to the corresponding soft constraints (for example, in a meta-constraint).

We can go further at the specification level by allowing the user to express more complex preferences. In [PRB00], a set of constraints on soft constraints, called *meta-constraints*, is introduced. Meta-constraints can be very helpful in the modelling process, as they allow the user to abstract to a higher level. With them we can limit the degree of violation of soft constraints, state certain homogeneity in the amount of violation of disjoint groups of constraints, etc. In [ABP+11] all these meta-constraints are implemented by translation into SMT. Unfortunately, with pseudo-Booleans this is much harder since we do not have integer decision variables.

Therefore, in the present work we have only implemented two meta-constraints, which allow to bound the amount of violation (i.e., the aggregated cost of the unsatisfied constraints) of a list of soft constraints:

- `maxCost(`ll`,`n`)` / `minCost(`ll`,`n`)`. Constraints the maximum/minimum aggregated cost of the unsatisfied constraints of each list of soft constraints in ll to be n.

We remark that ll is a list of lists of soft constraint labels. For instance, following the previous example we could constraint the amount of violation of nurse preferences to be 1. Notice that, this way, we would obtain more fair solutions.

```
MetaConstraints
  maxCost([[A,B],[C,D]],1);
```

3.2 Translation into pseudo-Booleans

For obtaining a pseudo-Boolean instance from a `Simply` model, the first two steps of the translation are the same as for generating an SMT instance: first, the meta-constraints are reformulated into a conjunction of constraints; next, the generators and the list comprehensions are unfolded, so that a conjunction of basic or global of constraints is obtained.

The first difference is with respect to linear arithmetic expressions, which are straightforwardly reformulated into pseudo-Boolean constraints. The translation of Boolean combinations (`Or`, `Implies`, ...) of basic constraints requires reification. This technique is also required to deal with weights when translating to the `pbo` format, and is described in the next subsection.

The translation of the `Sum`, `AtLeast` and `AtMost` global constraints is straightforward from its definition in Subsection 3.1. The translation of the meta-constraint `maxCost(`$[l_1,\ldots,l_m]$`,`n`)` results into a conjunction of `AtMost` global constraints, one for each list l_i of soft constraint labels. Each labelled soft constraint is reified into an auxiliary pseudo-Boolean variable, which is multiplied by the cost of the soft constraint, and those are the arguments given to the `AtMost` global constraints. The `minCost` meta-constraint is translated similarly but using the `AtLeast` global constraint. At the current stage of development, meta-constraints are still not implemented for the `wbo` format.

3.3 Reification

As explained in Section 2, pseudo-Boolean constraints are inequalities on linear combinations of 0-1 variables. Several types of constraints cannot be directly translated to pseudo-Boolean form, for instance the ones containing logical operators (such as `Or`, `Implies`, ...). To be able to express these kind of constraints we use reification, which means to reflect the constraint satisfaction into a 0-1 variable. Then, e.g., `a Or b` can be translated into $i_a + i_b \geq 1$, where i_a and i_b are the 0-1 variables corresponding to the Boolean variables `a` and `b`, respectively.

In addition, since soft constraints are not directly supported by the `pbo` format we use reification to manage their violation costs. To do that, each soft constraint is reified and a reverse reification variable is created using the equation: $reif\_var + reif\_var_{rev} = 1$ (note that in this reverse reification variable we are reflecting the falsification of the constraint). Then the reverse reification variable multiplied by the violation cost is added to the minimization function.

Reification is done according to the BigM method, as described in [Wil99]. In Example 4 we give an example.

Example 4 *Assume we want to reify a (possibly labelled) constraint* $\sum_j a_j x_j \leqslant b$*, being* δ *the corresponding reification variable. The relation between the constraint and* δ *can be modelled as follows:*

- $\delta = 1 \rightarrow \sum_j a_j x_j \leqslant b$ *is translated into* $\sum_j a_j x_j + M\delta \leqslant M + b$*, where* M *is an upper bound for the expression* $\sum_j a_j x_j - b$*.*
- $\delta = 0 \rightarrow \sum_j a_j x_j \not\leqslant b$*, i.e.,* $\delta = 0 \rightarrow \sum_j a_j x_j > b$*. Since none of the two standard pseudo-Boolean formats supports the* $>$ *operator, we need to rewrite the expression* $\sum_j a_j x_j > b$ *as* $\sum_j a_j x_j \geqslant b + \varepsilon$*, where* ε *can be taken as 1 as we are dealing with integers. The final resulting constraint is then* $\sum_j a_j x_j - (m - \varepsilon)\delta \geqslant b + \varepsilon$*, where* m *is a lower bound for the expression* $\sum_j a_j x_j - b$*.*

In the case that the reified constraint is soft, since pseudo-Boolean problems are minimization problems, it can be seen that the implication with $\delta = 0$ is unnecessary. However, our experimental results have shown it beneficial.

4 Experiments

Two WCSP have been chosen for testing the performance of the new system along its development: Soft BACP, a variant of the Balanced Academic Curriculum Problem, and the well known Nurse Scheduling Problem. We have chosen this two problems since they can be naturally encoded in a suitable pseudo-Boolean form. All experiments have been performed on a 2.8GHz 12GB Intel Core i7 system running GNU/Linux with kernel 2.6.35-32 (64 bits).

4.1 Solvers

The solvers used in the experiments, classified by families, are the following.

- Integer Programming based.

  SCIP (scip-2.1.1.linux.x86) - A mixed integer programming solver.

- Pseudo-Boolean resolution based.

  Sat4j (sat4j-pb-v20110329) - A Boolean satisfaction and optimization library for Java. With PBC, it uses a cutting planes approach or pseudo-Boolean resolution [Sat12].

  Clasp (clasp-2.0.6-st-x86-linux) - A conflict-driven answer set solver.

  Bsolo (beta version 3.2 - 23/05/2010) - A solver with cardinality constraint learning.

- Translation to MaxSAT or SMT based.

  PWBO (pwbo2.0) - Also known as the parallel wbo solver. Since the other considered solvers are single-threaded, we have not used its parallel feature. When pbwo is run with one thread it uses a unsat-based core algorithm [MML11].

x	c_1	c_2	c_3
p_1	1	0	0
p_2	0	1	0
p_3	0	0	1

x_aux	c_1	c_2	c_3
p_1	0	0	0
p_2	1	0	0
p_3	1	1	0

Table 1: SBACP prerequisites example.

PB/CT (pbct0.1.2) - Translates the problem into SAT modulo the theory of costs and uses the Open-SMT solver.

4.2 Soft BACP

The first problem we have considered is a relaxed version of the Balanced Academic Curriculum Problem (BACP, prob030 at http://www.csplib.org) where the prerequisite constraints are turned into soft constraints. Note that in this case it is possible to violate a prerequisite constraint between two courses but then, in order to reduce the pedagogic impact of the violation, we introduce a new hard constraint, the corequisite constraint, enforcing both courses to be assigned to the same period. We call this problem the *Soft Balanced Academic Curriculum Problem* (SBACP). The goal of the SBACP is to assign a period to every course, minimizing the total amount of prerequisite constraint violations, and satisfying the constraints on the number of credits and courses per period, and the corequisite constraints.

In Figure 3, we propose a modelling of the SBACP using `Max-Simply`. In order to obtain instances of the SBACP, we have over-constrained the BACP instances from the MiniZinc [NSB+07] repository, by reducing to four the number of periods, and proportionally adapting the bounds on the workload and number of courses of each period. With this reduction on the number of periods, we have been able to turn into unsatisfiable all these instances, hence there are no solutions with zero cost. In the model, `prerequisites[i,1]` denotes a course which is a prerequisite of the course denoted by `prerequisites[i,2]` for each `i`.[2]

In Table 1, we give a small example to illustrate the meaning of the two matrices of variables `x` and `x_aux` used in the model. We consider an instance with three courses and three periods. Matrix `x` shows in what period a course is made. Matrix `x_aux` shows when a course is already done, that is, it is filled with ones for the periods following the one in which the course is done; this also can be seen as the accumulated version of matrix `x`. This duality lets expressing prerequisites easily. Note that the first period row in matrix `x_aux` always contains zeros.

In Figure 4, we show how with the `maxCost` meta-constraint we can restrict the violations of prerequisites for each course to a number `n`. Note that, previously, we must associate the label `pre[i]` to the prerequisite constraints.

In Table 2, we show the performance results for the SBACP. We have tested the pseudo-Boolean version of `Max-Simply` with the different back-ends described in Section 4.1, along with the SMT version of `Max-Simply` using Yices, which is able to solve weighted SMT instances. The first group of rows reflect the execution times without the `maxCost` meta-constraint, while the second and third groups show the execution times with the meta-constraints `maxCost([...],1)` and `maxCost([...],2)`, respectively. Although not shown in the

[2] It is worth noting that in some constraints the $\geq$ operator could be replaced by the $=$ operator, but using $\geq$ results in a simpler formula when reifying; we have observed that this speeds up execution.

```
Problem:sbacp
  Data
    int n_courses; int n_periods; int n_prerequisites;
    int load_per_period_lb;       int load_per_period_ub;
    int courses_per_period_lb;    int courses_per_period_ub;
    int loads[n_courses];         int prerequisites[n_prerequisites, 2];

  Domains
    Dom pseudo = [0..1];

  Variables
    IntVar x[n_periods, n_courses]::pseudo;
    IntVar x_aux[n_periods, n_courses]::pseudo;

  Constraints
    % each course is done only once
    Forall(c in [1..n_courses]) { Sum([x[p,c] | p in [1..n_periods]], 1); };

    Forall(p in [1..n_periods]) {
    % for each period the number of courses is within bounds
      AtLeast([x[p,c] | c in [1..n_courses]], courses_per_period_lb);
      AtMost ([x[p,c] | c in [1..n_courses]], courses_per_period_ub);
    % for each period the course load is within bounds
      AtLeast([ loads[c] * x[p,c] | c in [1..n_courses]], load_per_period_lb);
      AtMost ([ loads[c] * x[p,c] | c in [1..n_courses]], load_per_period_ub); };

    % Prerequisites and Corequisites
    % if course c is done previously to period p then x_aux[p,c] = 1
    Forall(c in [1..n_courses]) {
      Forall(p in [1..n_periods]) {
        Sum([x[p2,c] | p2 in [1..p-1]], x_aux[p,c]); }; };

    % a course being a (soft) prerequisite of another cannot be done after the other
    Forall(i in [1..n_prerequisites]) {
      Forall(p in [1..n_periods-1]) {
        x_aux[p+1, prerequisites[i,1]] >= x[p, prerequisites[i,2]]; }; };

    % prerequisite violation has cost 1
    Forall(i in [1..n_prerequisites]) {
      Forall(p in [1..n_periods]) {
        (x_aux[p, prerequisites[i,1]] >= x[p, prerequisites[i,2]])@ {1}; };
    };
```

Figure 3: `Max-Simply` model for the SBACP.

```
    ...
    % prerequisite violation has cost 1
    Forall(i in [1..n_prerequisites]) {
      #pre[i]: Forall(p in [1..n_periods]) {
                (x_aux[p, prerequisites[i,1]] >= x[p, prerequisites[i,2]])@ {1}; }; };

    MetaConstraints
      maxCost([ [ pre[np] | np in [1..n_prerequisites], prerequisites[np,2] = c ]
       | c in [1..n_courses] ], n);
```

Figure 4: Meta-constraint for the SBACP problem.

		Max-Simply (PB)							Max-Simply (SMT)
		pwbo		Sat4j	clasp	bsolo	PB/CT	SCIP	Yices
		PBO	WBO						
without	Total	31.49	48.19	44.48	23.12	70.86	53.80	33.25	125.96
	Mean	1.50	2.41	1.59	0.83	2.53	2.15	1.19	4.50
	Median	0.05	0.04	0.99	0.32	0.91	1.14	1.10	1.61
	# t.o.	7	8	0	0	0	3	0	0
maxC 1	Total	5.37	*	14.37	0.64	0.85	8.00	6.60	8.29
	Mean	0.19	*	0.51	0.02	0.03	0.29	0.24	0.30
	Median	0.04	*	0.50	0.02	0.02	0.23	0.16	0.28
	# t.o.	0	*	0	0	0	0	0	0
maxC 2	Total	88.84	*	35.05	9.21	19.27	56.72	28.34	47.34
	Mean	6.35	*	1.25	0.33	0.69	2.03	1.01	1.69
	Median	0.07	*	0.98	0.29	0.35	1.39	0.72	1.24
	# t.o.	14	*	0	0	0	0	0	0

Table 2: SBACP solving times with a timeout of 60s. *: for now, meta-constrains are only available for the `pbo` format.

table, it is worth noting that in the case of using the constraint `maxCost([...],1)`, the problem becomes quite more constrained, resulting into 14 unsatisfiable instances out of the 28 instances. However, 9 of the 14 solutions of the remaining instances are improved in terms of fairness, since we have reduced the difference between the less violated course and the most violated course. Despite the fairness improvement, the mean cost of the solutions is penalized with an increment of 2.56 points. In the case of `maxCost([...],2)`, the problem becomes only slightly more constrained, transforming only one of the 28 instances in unsatisfiable. In this case, the number of improved instances, in terms of fairness, is only 2, with an increment in the mean cost of 1 point in the first case and of 2 points in the second case.

4.3 Nurse Scheduling Problem

The Nurse Scheduling Problem (NSP) is a classical constraint programming problem [WHFP95]. Each nurse has his/her preferences on which shifts wants to work. Conventionally a nurse can work in three shifts: day shift, night shift and late night shift. There is a required minimum of personnel for each shift (the shift cover). The problem is described as finding a schedule that both respects the hard constraints and maximizes the nurse satisfaction by fulfilling their wishes. In Figure 5 we propose a modelling of the NSP using `Max-Simply`, where we have the shift covers and the number of working days as hard constraints, and the nurse preferences as soft constraints. Notice that here, contrarily to the case of the SBACP, we have constraints with parametrized weights.

We have taken the instances from the NSPLib. The NSPLib is a repository of thousands of NSP instances, grouped in different sets and generated using different complexity indicators: size of the problem (number of nurses, days or shift types), shifts coverage (distributions over the number of needed nurses) and nurse preferences (distributions of the preferences over the shifts and days). Details can be found in [VM07]. In order to reduce the number of instances to work with, we have focused on a particular set: the N25 set, which contains 7920 instances. The N25 set has the following settings: 25 nurses, 7 days, 4 shift types (where the 4th is the free shift), a certain number of nurses required for every shift of each day and a value between 1 and 4

	Max-Simply (PB)			Max-Simply (SMT)
	pwbo		SCIP	Yices
	PBO	WBO		
# solved	5095	5064	7290	4088
# t.o.	2195	2226	0	3202
Mean	1.63	1.77	0.10	4.49
Median	0.38	0.68	0.09	1.52

Table 3: Results on the NSP with soft constraints on nurse preferences.

(from most desirable to less desirable) for each nurse, shift and day. The nurses are required to work exactly 5 days.

```
Problem:nsp
  Data
    int n_nurses; int n_days; int n_shift_types; int min_turns; int max_turns;
    int covers[n_days, n_shift_types]; int prefs[n_nurses, n_days, n_shift_types];

  Domains
    Dom pseudo = [0..1];

  Variables
    IntVar x[n_nurses, n_days, n_shift_types]::pseudo;

  Constraints
    % each nurse can only work one shift per day
    Forall(n in [1..n_nurses], d in [1..n_days]) {
      Sum([x[n,d,s] | s in [1..n_shift_types]], 1); };

    % minimum covers
    Forall(d in [1..n_days], s in [1..n_shift_types-1]) {
      AtLeast([x[n,d,s] | n in [1..n_nurses]], covers[d,s]); };

    % the number of free days is within bounds
    Forall(n in [1..n_nurses]) {
      If (min_turns = max_turns) Then {
        Sum([x[n,d,n_shift_types] | d in [1..n_days]], n_days - min_turns);
      } Else {
        AtLeast([x[n,d,n_shift_types] | d in [1..n_days]], n_days - max_turns);
        AtMost ([x[n,d,n_shift_types] | d in [1..n_days]], n_days - min_turns); }; };

    % penalize each failed preference
    Forall(n in [1..n_nurses], d in [1..n_days], s in [1..n_shift_types]) {
      (x[n,d,s] = 0) @ {prefs[n,d,s]}; };
```

Figure 5: Max-Simply model for the NSP.

Table 3 shows the summarized results for all the 7290 instances of the N25 set of the NSP. The solvers shown are the pseudo-Boolean version of Max-Simply with pwbo2.0 and SCIP, along with the SMT version of Max-Simply with Yices, as the rest of solvers have not been able to solve any instance within the time limit of 60 seconds. In these executions, SCIP excels, being able to solve all instances with a mean time which is an order of magnitude lower than the one of the second fastest solver.

5 Conclusions and future work

We have presented a work in progress variant of `Max-Simply` that takes advantage of the latest developments and improvements in the pseudo-Boolean solvers, giving the user a greater range of options when solving its constraint problems. Experiments presented here have shown that the pseudo-Boolean approximation can sometimes outperform SMT approximations. Even inside the pseudo-Boolean solvers ecosystem, depending on the solving strategies the results can vary greatly. As problems are easily encoded with a high-level language, the new system can also be seen as a tool to generate pseudo-Boolean benchmarks. It can allow developers and researchers to compare their different solvers and approaches. As for the future work, there are many interesting paths to follow:

- In problems where most decision variables are pseudo-Boolean and only a few are integer, it would be interesting to allow `Simply` to encode those integer variables using pseudo-Boolean variables. To this purpose, we should study the efficiency of different encodings.
- At the moment there are two meta-constraints implemented: `minCost` and `maxCost`. New meta-constraints should be introduced, enriching the language. This could be easier if integer variables were allowed.
- Another more ambitious point of interest should be developing and integrating a pseudo-Boolean theory solver in an SMT solver, in order to take advantage of both a DPLL search algorithm and specialized algorithms for the pseudo-Booleans.
- For now, the code generation phase is a bit slow, as no premature optimizations were made during the development phase of the tool. Now that we have obtained some promising results, code generation speed can and should be improved.

Acknowledgements: We thank Josep Suy for his helpful comments and suggestions.

Bibliography

[ABL09] C. Ansótegui, M. Bonet, J. Levy. Solving (Weighted) Partial MaxSAT through Satisfiability Testing. In *SAT 2009*. LNCS 5584, pp. 427–440. Springer, 2009.

[ABP+11] C. Ansótegui, M. Bofill, M. Palahi, J. Suy, M. Villaret. A Proposal for Solving Weighted CSPs with SMT. In *ModRef 2011*. Proceedings of the Tenth International Workshop on Constraint Modelling and Reformulation, pp. 5–19. 2011.

[BPSV09] M. Bofill, M. Palahi, J. Suy, M. Villaret. SIMPLY: a Compiler from a CSP Modeling Language to the SMT-LIB Format. In *ModRef 2009*. Proceedings of the Eigth International Workshop on Constraint Modelling and Reformulation, pp. 30–44. 2009.

[CK05] D. Chai, A. Kuehlmann. A fast pseudo-Boolean constraint solver. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 24(3):305–317, 2005.

[DM06] B. Dutertre, L. de Moura. The Yices SMT solver. Tool paper available at http://yices.csl.sri.com/tool-paper.pdf, August 2006.

[FHJ+08] A. Frisch, W. Harvey, C. Jefferson, B. Martínez-Hernández, I. Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints* 13(3):268–306, 2008.

[LS04] J. Larrosa, T. Schiex. Solving weighted CSP by maintaining arc consistency. *Artificial Intelligence* 159(1):1–26, 2004.

[MM02] V. Manquinho, J. Marques-Silva. Search pruning techniques in SAT-based branch-and-bound algorithms for the binate covering problem. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 21(5):505–516, 2002.

[MML11] R. Martins, V. Manquinho, I. Lynce. Exploiting Cardinality Encodings in Parallel Maximum Satisfiability. In *ICTAI 2011*. Proceedings of the 23rd IEEE International Conference on Tools with Artificial Intelligence, pp. 313–320. 2011.

[MMP09] V. Manquinho, J. Marques-Silva, J. Planes. Algorithms for Weighted Boolean Optimization. In *SAT 2009*. LNCS, pp. 495–508. Springer, 2009.

[NSB+07] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, G. Tack. MiniZinc: Towards a Standard CP Modelling Language. In *CP 2007*. LNCS 4741, pp. 529–543. 2007.

[PRB00] T. Petit, J. C. Regin, C. Bessiere. Meta-constraints on violations for over constrained problems. In *ICTAI 2000*. Proceedings of the 12th IEEE International Conference on Tools with Artificial Intelligence, pp. 358–365. 2000.

[RM09] O. Roussel, V. Manquinho. Pseudo-Boolean and cardinality constraints. *Handbook of Satisfiability* 185:695–733, 2009.

[RM12] O. Roussel, V. Manquinho. Pseudo-Boolean Competition. http://www.cril.univ-artois.fr/PB11/, 2011 (accessed 02/05/2012).

[Sat12] SAT4J Pseudo-Boolean Competition implementation. http://www.cril.univ-artois.fr/SAT09/solvers/booklet.pdf, 2009 (accessed 02/05/2012).

[VM07] M. Vanhoucke, B. Maenhout. NSPLib - a nurse scheduling problem library: a tool to evaluate (meta-) heuristic procedures. In *Operational research for health policy: making better decisions*. Proceedings of the 31st annual meeting of the working group on operations research applied to health services, pp. 151–165. 2007.

[WHFP95] G. Weil, K. Heus, P. Francois, M. Poujade. Constraint programming for nurse scheduling. *Engineering in Medicine and Biology Magazine, IEEE* 14(4):417–422, 1995.

[Wil99] H. P. Williams. *Model Building in Mathematical Programming*. Wiley, 4th edition, 1999.

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): Actas de las "XII Jornadas sobre Programación y Lenguajes (PROLE'2012), Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

Sesión 5

Fundamentos

Chair: *Dra. Paqui Lucio*

Sesion 5: Fundamentos
Chair: Dra. Paqui Lucio

Edelmira Pasarella, Fernando Orejas, Elvira Pino and Marisa Navarro. *Semantics of structured normal logic programs*.

Simone Santini. *Regular queries in event systems with bounded uncertainty*.

Pedro J. Morcillo, Gines Moreno, Jaime Penabad and Carlos Vázquez. *String-based Multi-adjoint Lattices for Tracing Fuzzy Logic Computations*.

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): Actas de las "XII Jornadas sobre Programación y Lenguajes (PROLE'2012), Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

Semantics of structured normal logic programs

Edelmira Pasarella[1], Fernando Orejas[1], Elvira Pino[1], Marisa Navarro[2] *

[1] Departament de LSI, Universitat Politècnica de Catalunya, Jordi Girona, 1-3. 08034 Barcelona, Spain
[2]Departamento de LSI, Universidad del País Vasco, Paseo Manuel de Lardizabal, 1, Apdo 649, 20080 San Sebastián, Spain

Abstract: In this paper we provide semantics for normal logic programs enriched with structuring mechanisms and scoping rules. Specifically, we consider constructive negation and expressions of the form $Q \supset G$ in goals, where Q is a program unit, G is a goal and $\supset$ stands for the so-called embedded implication. Allowing the use of these expressions can be seen as adding block structuring to logic programs. In this context, we consider static and dynamic rules for visibility in blocks. In particular, we provide new semantic definitions for the class of normal logic programs with both visibility rules. For the dynamic case we follow a standard approach. We first propose an operational semantics. Then, we define a model-theoretic semantics in terms of ordered structures which are a kind of intuitionistic Beth structures. Finally, an (effective) fixpoint semantics is provided and we prove the equivalence of these three definitions. In order to deal with the static case, we first define an operational semantics and then we present an alternative semantics in terms of a transformation of the given structured programs into flat ones. We finish by showing that this transformation preserves the computed answers of the given static program.

Keywords: semantics, normal logic programs, embedded implication, visibility rules, structuring mechanism, intuitionistic structures

1 Summary

Regarding dynamic normal logic programs, the intuition of the interpretation of embedded implications is that, given a program P, to prove the query $Q \supset G$ it is necessary to prove G with the program $P \cup Q$. This is formalized in a seminal work by Miller, using the inference rule shown in Figure 1. In Miller's approach, both implications, clausal implication and embedded implication,

$$\frac{P \cup Q \vdash_{dyn} G}{P \vdash_{dyn} Q \supset G}$$

Figure 1: Dynamic rule

* This work has been partially supported by the Spanish CICYT project FORMALISM (ref. TIN2007-66523) and by the AGAUR Research Grant ALBCOM (ref. SGR 20091137)

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): PROLE'2012, pp. 159-160, ISBN:978-84-15487-27-2.
Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

are interpreted as intuitionistic implications. In particular, a main result of this work is that the proof-theoretic semantics for this kind of programs can be given in terms of intuitionistic logic. Moreover, he proposed a model-theoretic semantics in terms of Kripke models. In our paper we extend this approach to the case of programs that also include negation. First, we introduce an operational semantics for the class of normal logic programs with embedded implication. This semantics is an extension of constructive negation with the above rule to handle implication goals. Extending similarly the model-theoretic semantics, both to define the logical semantics and the least fixpoint semantics of a program, is quite more involved. In particular, a main difficulty comes from the non-monotonic nature of negation in logic programming which does not fit well with modularity. We have the intuition that both connectives could have a natural and reasonably simple semantics in terms of intuitionistic (Beth) models and this semantics would make more explicit the intuitionistic nature of negation in logic programming already pointed out by other authors. Therefore, we define the Beth models that capture our intuition together with their associated forcing relation. Then, we introduce an immediate consequence operator showing that it is monotonic and continuous. Moreover, we show that the least fixpoint of this operator coincides with the least model of the given program. Finally, the operational semantics is proved to be sound and complete with respect to the least fixpoint semantics.

To deal with static normal logic programs, we consider the same kind of programs as in previous case, but interpreting embedding with static scoping. The rule for handling implications in this case is shown in Figure 2. The intuition of the notation $P|Q \vdash_{st} G$ is that we want to solve G in

$$\frac{P|Q \vdash_{st} G}{P \vdash_{st} Q \supset G}$$

Figure 2: Static rule

a scope consisting of two nested blocks of definitions, P and Q, where Q is *local* to P, i.e. $P|Q$ represents this kind of block inclusion. To solve the goal G in the scope of $P|Q$ we can use any definition that is present in either P or Q. This is just as in block-structured procedural languages where, for solving a reference in a given scope, we may use any visible definition from any of the blocks which are global to that reference. To define a semantics to this class of programs, instead of developing a new framework to define a model-theoretic semantics, we show how these programs can be transformed into standard normal programs. Moreover, we prove that this translation is sound and complete with respect to the operational semantics of the extended programs. In addition, it must be pointed out that this transformation is easy to implement, which means that we can easily build this kind of extension on top of a standard logic programming language. This approach has been used in previous work to deal with positive propositional static programs and to translate modal logic programs with embedded implication into Horn programs. To achieve our goal, we first introduce an operational semantics for the class of static normal logic programs with embedded implications. Then, we present the transformation semantics and, finally, we prove the soundness and completeness of the transformation.

Regular queries in event systems with bounded uncertainty

Simone Santini*

Escuela Politécnica Superior
Universidad Autónoma de Madrid

simone.santini@uam.es

Abstract: In this paper we consider the problem of matching sequences of events against regular expressions when the detection of atomic events is subject to bounded time uncertainty. This work represents an extension of previous work, in which the semantics of matching continuous streams of events was defined for precisely placed atomic events.

In this paper we show that the general problem of matching with time uncertainty is, under a reasonable semantics, NP-complete. However, the problem is polynomial if the expression is fixed.

Keywords: event systems, uncertainty, regular expressions, semiorders

1 Introduction

In this paper we consider the problem of matching sequence of events against regular expressions when the detection of atomic events is subject to bounded time uncertainty. This work represents an extension of [San10], in which the semantics of matching continuous streams of events was defined for events with no time uncertainty.

In this paper we show that the general problem of matching with time uncertainty (viz. the problem with unrestricted expression and unbounded alphabet) is, under a reasonable semantics, NP-complete. However, the problem is polynomial if the expression is of fixed complexity; we present a constructive proof by exhibiting a polynomial algorithm for matching with uncertainty and proving its correctness.

The assumption of bounded uncertainty is a reasonable one in cases like video detection, in which the inherent imprecision of the analysis algorithm and/or the event definition makes an exact localization impossible.

2 Uncertainty and semiorders

Definition 1 An *event* is a triple $(a, \xi, [t_e, t_e + \Delta])$, where $a \in \Sigma$ (Σ being finite) is the *type* of the event, ξ are its *parameters*, and its occurrence time belongs to the interval $[t_e, t_e + \Delta]$, Δ being a system-wide uncertainty constant.

* This work was supported by the *Ministerio de Educación y Ciencia* under the grant N. TIN2011-28538-C02, *Novelty, diversity, context and time: new dimensions in next-generation information retrieval and recommender systems.*

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): PROLE'2012, pp. 161-175, ISBN:978-84-15487-27-2.
Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

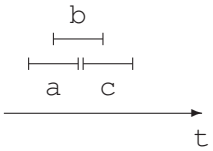
Remark 1 *The parameter ξ contains information that gives events their identity (so that, for example, two distinct events of the same type can occur at the same time). Events are matched based on their type, so parameters will in general be omitted. Also, the unit of time can be defined in such a way that $\Delta = 1$. This choice we shall make, leading to the representation $e = (a, [t_e, t_e + 1])$. We shall use the notation $\lambda(e)$ and $\tau(e)$ to indicate the type and the first extremum of the time interval for an event, that is, $\lambda(a, [t_e, t_e + 1]) = a$ and $\tau(a, [t_e, t_e + 1]) = t_e$.*

Definition 2 Given a set of events E, the relations $\prec \subseteq E \times E$ and $\| \subseteq E \times E$ are defined as

$$\begin{aligned} \prec &= \{(e, e') | \tau(e) + 1 < \tau(e')\} \\ \| &= \{(e, e') | (e, e') \notin \prec \text{ and } (e', e) \notin \prec\} \end{aligned} \tag{1}$$

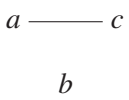
We shall commonly write $e \prec e'$ and $e \| e'$ in lieu of $(e, e') \in \prec$ and $(e, e') \in \|$, respectively.

The relation $\prec$ is antisymmetric ($x \prec y \Rightarrow \neg(y \prec x)$) and transitive ($x \prec y \wedge y \prec z \Rightarrow x \prec z$), but it does not form a complete disjunction. That is, the events form a partially ordered set (poset) $(E, \prec)$. The relation $||$ is symmetric, but not necessarily transitive. In the following example



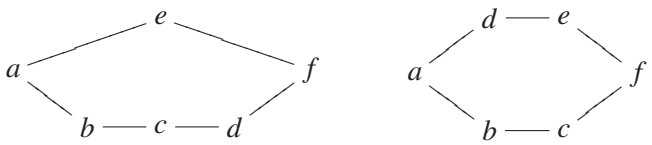
it is $a||b$ and $b||c$, but $a \prec c$.

Remark 2 *In this paper, from now on, we shall represent posets of events following the time ordering convention rather than the standard poset one. If $x \prec y$, x will be placed to the left of y, and the two will be united by a line. So, we shall represent the previous poset as*



Definition 3 The relation $\preceq$ is defined as $x \preceq y \equiv (x \prec y \vee x = y)$; $\preceq$ is reflexive, antisymmetric and transitive.

The fact that the uncertainty intervals are unitary, however, limits the structure of the posets that can be expected in event systems. In particular, it is known that unitary interval orders generate a class of posets known as *semiorder* [PV97]. Semiorders can be defined as the class of partial orders in which no restriction is isomorphic to one of the following two graphs

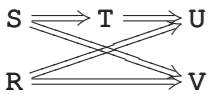


It is easy to check that events with unitary intervals will never generate any of these orders. Consider the first partial order above. Since $b \prec c \prec d$, it is $b + 1 < d - 1$, therefore e can overlap b or d, but not both. This leads to a computational problem simpler than the general interval algebra of [All83].

Remark 3 *Semi-orders can be represented as sequences of blocks with similar structure. In the following, capitalized latin letters will indicate* layers, *that is, sets of events such that if* $e, e' \in \mathtt{A}$, *then* $e \| e'$. *The notation*

$$\mathtt{A} \Longrightarrow \mathtt{B}$$

will indicate a complete precedence of the events in A *over the events in* B, *that is, for all* $e \in \mathtt{A}$ *and* $e' \in \mathtt{B}$, *it is* $e \prec e'$. *A semi-order is then a sequence of blocks with structure*



(2)

where T *may be empty.*

3 Semantics of matching

Our problem is to find out whether a collection of events with a precedence relation with the structure of a semi-order *matches* a regular expression with variables ϕ. In order to make these concepts more precise, we need a definition that applies to general partial orders.

Definition 4 Given a partial order $(E, \prec)$, its *linearization* is a totally ordered set $(E, <)$ where, for each $e_1, e_2 \in E$, $e_1 \prec e_2 \Rightarrow e_1 < e_2$.

In other words, the linearization of a partial order is any total order that we obtain by extending the relation $\prec$ giving an arbitrary order to non comparable elements.

With the events laid out as partial orders, we can consider a *strong* or a *weak* semantics. Informally speaking, strong semantics considers as sequential only events that are sequential in the graph, that is, e_1 precedes e_2 only if $e_1 \prec e_2$. If $e_1 || e_2$ there is not enough information for the determination, and we consider the events as happening at the same time. In *weak* semantics, on the other hand, given two events $e_1 || e_2$ we consider that the two can be ordered in any way, and that each of the permissible orders can be a match for the regular expression.

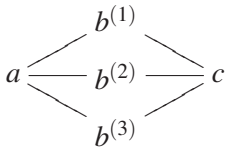
Definition 5 Given a partial order $(E, \prec)$ and a regular expression ϕ, $(E, \prec)$ matches ϕ in the strong sense ($E \models^{\mathrm{S}} \phi$) if there exists a total order $(E', <)$, with $E' \subseteq E$ and $\prec \subseteq <$ such that $(E', <)$ matches ϕ in the standard sense, i.e. there exists a set $E' \subseteq E$ such that, if $E' = \{e_1, \ldots, e_n\}$ and $e_1 < e_2 < \cdots < e_n$, then $\lambda(e_1)\lambda(e_2)\cdots\lambda(e_n) \models \phi$ in the standard semantics of regular expressions.

Moreover, E' is a time-maximal subset of E, that is, there are $e, e' \in E'$ such that for all $e \in E$ it is neither $e \prec e'$ nor $e'' \prec e$.

Definition 6 Given a partial order $(E, \prec)$ and a regular expression ϕ, $(E, \prec)$ matches ϕ in the weak sense ($E \models^{\mathrm{W}} \phi$) if there exists a total order $<$ with $< \subseteq \prec$ (i.e. a linearization of $(E, \prec)$) that matches ϕ in the standard sense as given in the previous definition.

Remark 4 *The strong semantics is not terribly interesting neither from a computational nor from an application point of view. It implies that there is a path in the semi-order graph that*

matches the regular expression. We can find this path using the standard algorithms for processing regular expressions [San12]. Weak semantics is more interesting, since it entails that, while matching, we can take non-comparable events in any order that we want. Consider the expression ab^*c and the event graph



(We use an apex in parentheses to distinguish different instances of the same event type; formally the apex is a component of the parameters ξ of the event. Here $b^{(1)}$, $b^{(2)}$, and $b^{(3)}$ are all events of type b, which match the symbol b in the expression.)

In the strong semantics, the only sequences of events that match the expression are the three linear orders $ab^{(1)}c$, $ab^{(2)}c$, $ab^{(3)}c$. In the weak semantics, however, we are free to order the three b events as we please, since we don't know in what order they are actually appearing, so we have six possible solutions corresponding to the six permutations of $\{1,2,3\}$, from $ab^{(1)}b^{(2)}b^{(3)}c$ through $ab^{(3)}b^{(2)}b^{(1)}c$.

4 Complexity

The previous remark shows that finding *all* the linearizations of a semi-order that match an expression can be computationally intractable, since their number can be exponential in the size of the event set. In the remark the problem arises because there is a set of $O(n)$ incomparable events, but theorem 4.1 in [San12] shows that we can expect intractability in graphs with incomparable sets containing as little as two events (in this case, however, there will be $O(n)$ sets of incomparable events). In most applications, however, it is not necessary to find all linearizations: given a set E of events and an expression ϕ we want to know, most of the time, whether the events of E can be construed, without violating any established precedence constraint, as a sequence that matches ϕ. We are, in other words, interested in the solution of the following problem:

> SEMI-ORDER-MATCH(n,m): given a semi-order $(E,\prec)$, with $|E| \leq n$, and an expression ϕ with $|\phi| \leq m$[1], determine whether $(E,\prec) \models^{\mathrm{W}} \phi$, that is, whether there exists a linearization of $\prec$ such that $(E,<) \models \phi$.

Solving SEMI-ORDER-MATCH(n,m) is inherently hard. We shall prove it by considering the following problem:

> MULTI-SET-MATCH(n,m): given a finite alphabet Σ, a bag N with n elements of Σ (with repetitions), and a regular expression ϕ, with $|\phi| \leq m$, is it possible to use the elements of N to form a string that matches ϕ?

If we consider the elements of N as events, then N is a special case of a semi-order.

[1] For our purposes, we can interpret $|\phi|$ as the number of symbols in ϕ.

Theorem 1 MULTI-SET-MATCH(n,m) *is NP-complete*

Proof. We shall prove the theorem by reduction from X3S(n) (Exact cover by 3-sets, [GJ79]). Let $Q = \{a_1, \ldots, a_n\}$ be given, with $n = 3q$, and let $\{C_1, \ldots, C_k\}$ be subsets of Q.

We build an alphabet $\Sigma = \{a_1, \ldots, a_n, \#_1, \ldots, \#_k\}$, where $\#_1, \ldots, \#_k$ are unique elements not contained in Q. Our set N consists of all the elements of the alphabet, repeated only once, i.e. $N = \Sigma$. For every subset $C_i = \{a_{i_1}, a_{i_2}, a_{i_3}\}$ we shall build the expression

$$\phi_i = a_{i_1}\, a_{i_2}\, a_{i_3}\, \#_i \tag{3}$$

The regular expression that we shall consider is

$$\phi = \phi_1^* \cdots \phi_k^* \#_1^* \cdots \#_k^* \tag{4}$$

We claim that X3S has a solution if and only if MULTI-SET-MATCH(n,m) has a solution with the given set and expression.

Assume first that MULTI-SET-MATCH(n,m) has a solution. First note that each expression ϕ_i is matched at most once since in order to match it we have to use the corresponding symbol $\#_i$, and that symbol appears only once in N. So, once we have matched a ϕ_i there are no more symbols in N that will allow us to match it again. If the expression ϕ_i is matched zero times, then (and only then) the corresponding expression $\#_i^*$ will be matched once to use the symbol $\#_i$. If ϕ is matched, then the expressions ϕ_i will use up all the elements of N, and use up each element only once, since each a_i appears only once in N.

If X3S(n) has a solution, let $C_{u_1}, \ldots, C_{u_q}$ be the subsets that cover Q. Then the set N matches ϕ by taking once the expressions $\phi_{u_1}, \ldots, \phi_{u_q}$ and taking once the $\#_i$ expressions corresponding to the # symbols that have not been used.

On the otehr hand, it is easy to see that MULTI-SET-MATCH(n,m) is NP: a nondeterministic algorithm simply has to guess the correct ordering of the elements of the multi-set and then use the standard regular expression algorithm (which is poilynomial) to verify whether the order matches the expression. □

Remark 5 MULTI-SET-MATCH(n,m) *is equivalent to determining that* E *belongs to the Parikh image of* ϕ *[Par66] so the previous theorem is also an alternative proof–simpler and more self-contained than the one given in [KT10]–that such problem is NP-complete. The previous proof assumes an unbounded alphabet, as its size grows linearly with* n*. If we don't allow this to happen, that is, if we fix the size of* Σ*, then the problem is polynomial in* n *(but exponential in* $|\Sigma|$*, a consequence of theorem 8 of [KT10]).*

Theorem 2 SEMI-ORDER-MATCH(n,m) *is NP-complete.*

Proof. The problem can be seen to be NP using the same argument as in the previous theorem. MULTI-SET-MATCH(n,m) is a special case of SEMI-ORDER-MATCH(n,m), since a multi-set is a semi-order in which the relation $\prec$ is empty, so the trivial reduction from SEMI-ORDER-MATCH(n,m) proves that this problem is NP-complete as well. □

Corollary 1 MULTI-SET-MATCH(n,m) *stays NP-complete even if the expression* ϕ *contains no operator* $+$.

Proof. The expression that we used to prove theorem 1 does not contain any + operator. □

Remark 6 *From now on, we shall only consider conjunctive expressions, that is, expressions without "+" operators. If this operation is present in an expression, we shall assume that the expression is put in disjunctive normal form, and that teh various disjunctions are analyzed in parallel.*

5 Fixed regular epression

If we consider fixed regular expressions, that is, if we don't consider the size of the regular expression as part of the variable size of the problem, then we can match the expression in polynomial time. We shall indicate with MULTI-SET-MATCH(n) and SEMI-ORDER-MATCH(n) the problems with fixed expressions, in which the only parameter is the size of the event set, n. We shall illustrate the idea with an example.

Example:
Consider the expression

$$\phi = (a(ab)^*bb)^*cc^*ab \tag{5}$$

and a multi-set

$$N = \{\!\{a,a,a,a,b,b,b,b,b,c,c,c\}\!\} = \{\!\{(a,4),(b,5),(c,3)\}\!\} \tag{6}$$

In order to match the expression ϕ with the elements of N, we have only to decide how many times we repeat each one of the expressions under the star. We don't know this a priori, so we shall introduce three integer paramenters m_1, m_2, m_3 that tell us how many times we match the expressions (ab), (abb), and c respectively. Note that although (ab) is contained into $(a(ab)^*bb)$, the two are considered as independent: not every time that we match (abb) do we have to match the internal cycle (ab) the same number of times. The string

$$a\,abab\,bb\,abb\,cab \tag{7}$$

matches (abb) twice: the first time it also matches (ab) twice, the second time it doesn't match (ab) at all. Each expression, matched a given number of times, "consumes" a corresponding number of elements of N. So, if we match (ab) m_1 times, we need $n_a^1 = m_1$ symbols a, $n_b^1 = m_1$ symbols b, and $n_c^1 = 0$ symbols c. Similarly, if we match $(a--bb)$ m_2 times (we do not count again the matches of the internal loop (ab), since we have already counting them with the parameter m_1), we need $n_a^2 = m_2$ symbols a, $n_b^2 = 2 \cdot m_2$ symbols b, and $n_c^2 = 0$ symbols c. Finally, for the expression c^* we need $n_a^3 = n_b^3 = 0$ and $n_c^3 = m_3$. In order to match the whole expression with the specified number of cycles, we need

$$\begin{array}{rcllll} n_a & = & n_a^1 + n_a^2 + n_a^3 + 1 & = & m_2 + m_1 + 1 & \text{symbols } a \\ n_b & = & n_b^1 + n_b^2 + n_b^3 + 1 & = & 2m_2 + m_1 + 1 & \text{symbols } a \\ n_c & = & n_c^1 + n_c^2 + n_c^3 & = & m_3 + 1 & \text{symbols } a \end{array} \tag{8}$$

What are the ranges of these paramenters? Each sub-expression under a star uses at least one symbol for each iteration, so no solution can exist if $m_i > |N|$ for some i, which leads to the limits $0 \leq m_i \leq |N|$. An additional restriction is that if the parameter of a star has value 0 (the corresponding sub-expression is never matched), then the parameters corresponding to all sub-expressions under that star must also have value 0.

From (8) we can now derive a system of diophantine equations by equating the number of symbols that we need in order to match ϕ with the symbols that we have available in N:

$$\begin{aligned} m_2 + m_1 + 1 &= 4 \\ 2m_2 + m_1 + 1 &= 5 \\ m_3 + 1 &= 3 \end{aligned} \tag{9}$$

which has a solution for $m_1 = 2$, $m_2 = 1$, and $m_3 = 2$. So it is possible to match ϕ using the elements of N. For example, we can create the string

$$aababbbcab. \tag{10}$$

* * *

In the general case, given the alphabet $\Sigma = \{a_1, \ldots, a_k\}$, an expression ϕ, and a multi-set $N = \{\!\{(a_1, q_1) \ldots, (a_k, q_k)\}\!\}$, let n^i_ϕ be the number of occurrences of a_i in a string that matches ϕ. We use the variables $\phi', \phi'', \ldots$ to indicate sub-expressions of ϕ, and $\psi', \psi'', \ldots$ to indicate sub-expressions (possibly empty) in which the star does not appear. We can then calculate n^i_ϕ recursively, based on the structure of ϕ, as follows:

$$\begin{aligned} \phi \equiv a_i &\Rightarrow n^i_\phi = 1; j \neq i \Rightarrow n^j_\phi = 0; \\ \phi \equiv \phi'\phi'' &\Rightarrow n^i_\phi = n^i_{\phi'} + n^i_{\phi''} \\ \phi \equiv (\psi'\phi'^*\psi'')^* &\Rightarrow n^i_\phi = m_\phi(n^i_{\psi'} + n^i_{\psi''}) + m_{\phi'} n^i_{\phi'} \end{aligned} \tag{11}$$

where, in the last relation, m_ϕ and $m_{\phi'}$ are new parameters not used in any other derivation with $0 \leq m_\phi, m'_\phi \leq |N|$. Finally, we write the system of equations:

$$n^i_\phi = q_i \tag{12}$$

This is a l system of linear diophantine equations in the variables $m^1_\phi, \ldots, m^k_\phi$, where l is the number of star operators in the expressions. We can trivially find all solutions by checking all possible $|N|^l$ combinations of the values. The condition on the nested stars (if $m^i_\phi = 0$ for an expression ϕ^*, then $m^j_{\phi'} = 0$ for all j and for all sub-expressions ϕ' of ϕ) can easily be checked in time at most l.

* * *

The result is less picayune that it might appear. It is true that proving that MULTI-SET-MATCH(n) is polynomial doesn't prove that SEMI-ORDER-MATCH(n) is, but we can use this result to build a polynomial algorithm for SEMI-ORDER-MATCH(n). First, note that MULTI-SET-MATCH(n) is a special case (restricted to multi-sets) of weak modeling, so MSM returns true if and only if $N \models^{\mathrm{W}} \phi$. Second, we introduce the concepts of the *truncate* of an expression and of its *continuation*.

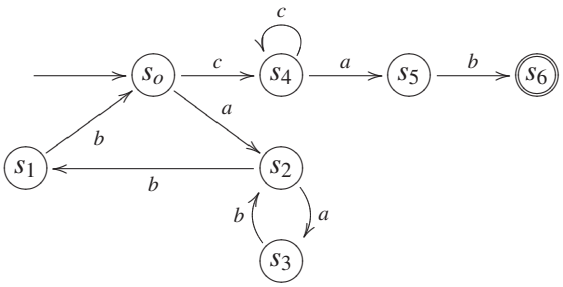
Definition 7 Let $M = (\Sigma, S, \delta, s_0, F)$ be a finite state automaton, where Σ is the input alphabet, S the set of states, $\delta : \Sigma \times S \to S$ the (deterministic) state transition function, $s_0 \in S$ the initial state, and $F \subseteq S$ the set of final states. Let $s \in S$; the *s-truncate* of M, $M[\to s]$, is the automaton

$$M' = (\Sigma, S, \delta, s_0, \{s\})$$

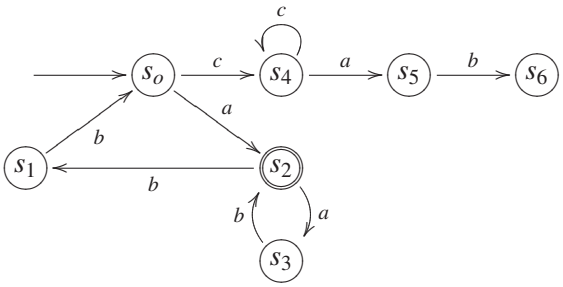
We indicate with $L(M)$ the language recognized by an automaton, and with $L(\phi)$ the language generated by a regular expression. If s is not reachable from s_0, then it is $L(M[\to s]) = \emptyset$.

Definition 8 Let ϕ be a regular expression, and M an automaton that recognizes ϕ. The *s-truncate* of ϕ, $\phi[\to s]$ is the expression recognized by the automaton $M[\to s]$.

Example:
Consider again the expression (5), which is recognized by the automaton M_ϕ:



The s_2 truncate of the expression, (5) is the expression recognized by the automaton $M_\phi[\to s_2]$:



that is, the expression

$$\phi[\to s_2] \equiv (a(ab)^*bb)^*a(ab)^* \tag{13}$$

* * *

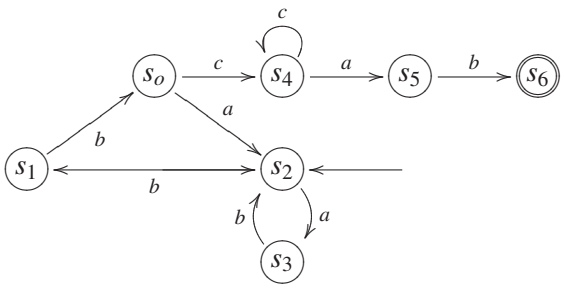
Note that the truncate of an expression is not, in general, a sub-expression in the syntactic sense. Also, if we indicate with $|M|$ the number of states of M, there are exactly $|M|$ truncates of M, so the number of distinct truncates of an expression ϕ is $O(|\phi|)$.

Definition 9 Let $M = (\Sigma, S, \delta, s_0, F)$ be a finite state automaton, and $s \in S$. The *s-suffix* of M is the automaton

$$M[s\rightarrow] = (\Sigma, S, \delta, s, F)$$

The s-suffix of an s-truncate of an automaton, $M[\rightarrow s\rightarrow] \equiv M[\rightarrow s][s\rightarrow]$, will be usually referred to as the *continuation* of the truncate. Also, define $M[s_1 \rightarrow s_2] \equiv M[s_1\rightarrow][\rightarrow s_2]$ The suffix and the continuation of an expresion ϕ, $\phi[s\rightarrow]$, $\phi[\rightarrow s\rightarrow]$, and $\phi[s_1 \rightarrow s_2]$ are defined analogously.

Example:
The s_2-suffix of the previous automaton is the automaton



and the s_2-continuation of ϕ is

$$\phi[s_2\rightarrow] \equiv (ab)^*bb(a(ab)^*bb)^*cc^*ab \tag{14}$$

* * *

We can compose these operators by taking further truncations of this continuation, and so on. If L_1 and L_2 are languages in Σ^*, define

$$L_1 \cdot L_2 = \{\omega_1 \cdot \omega_2 | \omega_1 \in L_1 \ \wedge \ \omega_2 \in L_2\} \tag{15}$$

(in particular, if $L_1 = \emptyset$ or $L_2 \in \emptyset$, then $L_1 \cdot L_2 = \emptyset$).

Lemma 1 *Let $M = (\Sigma, S, \delta, s_0, F)$, and $s \in S$. Then*

$$L(M[\rightarrow s]) \cdot L(M[s\rightarrow]) \subseteq L(M) \tag{16}$$

Proof. Let $\omega \in L(M[\rightarrow s]) \cdot L(M[s\rightarrow])$; then $\omega = \omega_1 \cdot \omega_2$, with $\omega_1 \in L(M[\rightarrow s])$ and $\omega_2 \in L(M[s\rightarrow])$. Therefore, if we give ω_1 as an input to M, M will go from the initial state to the state s and, if we apply ω_2 to M in state s, it will move to a state $s' \in F$. So, the whole string ω leads M from s_0 to $s' \in F$, therefore $\omega \in L(M)$. □

Note that the reverse is not true. In the previous example, $cab \in L(M)$, but $cab \notin L(M[\rightarrow s_2]) \cdot L(M[s_2\rightarrow])$, since while recognizing *cab* the automaton M never goes through the state s_2. The following lemma applies to the syntactic form of the truncates, and its proof is straightforward:

Lemma 2 *Let ϕ be an expression recognized by an automaton M, and $s, s' \in S$. Then*

$$\phi[\rightarrow s]\phi[s \rightarrow s'] \equiv \phi[\rightarrow s'] \tag{17}$$

That is, the syntactic juxtaposition of a truncate of an expression and a truncate of its continuation is still a truncate of the original expression.

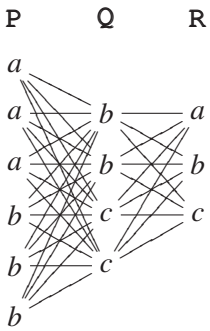
* * *

In order to see how truncates and their continuations can solve matching in polynomial time (given a polynomial algorithm for MULTI-SET-MATCH(n)), we shall begin with an example.

Example:
Consider again the expression

$$\phi \equiv (a(ab)^*bb)^*cc^*ab$$

and the semi-order



(18)

where, for the sake of reference, we have indicated the three layers as P, Q, and R. The events of the layer P take place, in an undetectable order, before all the events of layers Q and R. So, if the whole structure matches ϕ, it must be possible to organize the events of P in such a way that they constitute a prefix of a string that matches ϕ. That is, P must be ordered in such a way that there is a truncate $\phi[\rightarrow s_1]$ of ϕ and $P \models^{\mathrm{W}} \phi[\rightarrow s_1]$. Applying the MULTI-SET-MATCH(n) algorithm to all possible truncates (there are at most $|\phi|$), we find that P matches

$$\phi[\rightarrow s_1] \equiv \underbrace{(a\underbrace{(ab)^*}_{m_1}bb)^*a}_{m_2} \tag{19}$$

with $m_1 = m_2 = 1$ (with both expressions under the star being repeated once; the matching string is *aabbba*). The rest of the structure must match the continuation of the expression, so we have reduced the problem to matching

$$\phi[s_1 \rightarrow] \equiv (ab)^*bb(a(ab)^*bb)^*cc^*ab \tag{20}$$

against the semi-order

$$\begin{array}{cc} \mathtt{Q} & \mathtt{R} \\ b & a \\ b & b \\ c & c \\ c & \end{array} \tag{21}$$

Now we repeat the algorithm to find that Q matches

$$\phi[\rightarrow s_1 \rightarrow s_2] \equiv \underbrace{(ab)^*}_{m_1} bb \underbrace{(a\underbrace{(ab)^*}_{m_2} bb)^*}_{m_3} c \underbrace{c^*}_{m_4} \tag{22}$$

with $m_1 = m_2 = m_3 = 0$ and $m_4 = 1$ (string $bbcc$). The continuation is now

$$\phi[s_2 \rightarrow] \equiv c^* ab \tag{23}$$

and R matches it, showing that the whole semi-order matches the expression ϕ.

A possible complication is that a set of events (viz. a layer in a semi-order) can match several truncates of an expression. Notice, however, that in spite of this multiplicity of matching possibilities there can be no combinatorial explosion in the number of possible matching as we advance into the layers of the semi-order, as the total number of truncates that have to be examined is limited by lemma 2: it is at most equal to the number of states in the automaton that recognizes ϕ (since we assume that ϕ contains no "+", this number is linear in $|\phi|$). In the following, for the sake of simplicity, we shall always consider that a set of events matches at most one truncate. In so doing, we underestimate the complexity of the problem at most by a factor $|\phi|$, so any result pertaining to polynomiality will not be invalidated.

Before delving into the details of the polynomial algorithm we shall consider a special case of MULTI-SET-MATCH(n) on which the algorithm is be based. Suppose we have *two* multi-sets

$$\begin{aligned} U &= \{\!\{(a_i,u_1),\ldots,(a_k,u_k)\}\!\} \\ V &= \{\!\{(a_i,v_1),\ldots,(a_k,v_k)\}\!\}. \end{aligned} \tag{24}$$

We want to find a sub-set $V' \subseteq V$ such that $U \cup V' \models^{\mathtt{W}} \phi$. We can solve this problem without looking at all the subsets of V (that is, without combinatorial explosion) by noting that the problem has a solution if and only if the diophantine system (12) has a solution $(m_1,\ldots,m_k)$ with $u_i \leq m_i \leq u_i + v_i$. If such a solution exist, then we define the set

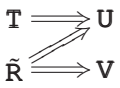
$$\tilde{V} = V - V' = \{\!\{(a_i, v_1 + u_1 - m_1),\ldots,(a_k, v_k + u_k - m_k)\}\!\}. \tag{25}$$

It is easy to see that $\tilde{V} \subseteq V$.

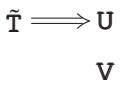
* * *

We now turn to the problem of semi-order matching.The most general substructure that we can find in a semi-order is that of (2), where T may be empty. As we did before, here we indicate with rectangles groups of events for which the order can't be determined, and the arrows imply a full connection between the corresponding layers. So, for example, all events in S come before all the events in T. The algorithm S-MATCH reduces matching on this structure to matching on a layer in three steps.

i) Find $R' \subseteq R$ and a state s_1 such that $S \cup R' \models^{\mathrm{W}} \phi[\rightarrow s_1]$; call $\tilde{R} = R - R'$. We now have to match $\phi[s_1 \rightarrow]$ to



ii) find $T' \subseteq T$ and a state s_2 such that $\tilde{R} \cup T' \models^{\mathrm{W}} \phi[\rightarrow s_1 \rightarrow s_2]$; set $\tilde{T} = T - T'$; the problem is now reduced to matching the continuation $\phi[s_2 \rightarrow]$ to



iii) find $V' \subseteq V$ and a state s_3 such that $\tilde{T} \cup V' \models^{\mathrm{W}} \phi[\rightarrow s_1 \rightarrow s_2 \rightarrow s_3]$; set $\tilde{V} = V - V'$; the problem is now reduced to matching the continuation $\phi[s_3 \rightarrow]$ to

If what remains is the last layer of the semi-order, then we apply the algorithm MULTI-SET-MATCH(n), otherwise we repeat steps i)–iii) with the semi-order that begins with this new layer. The algorithm requires $O(n|\phi|)$ applications of MULTI-SET-MATCH(n) and is therefore PTIME.

5.1 Correctness

In order to show that the algorithm is correct, we shall consider a class of series-parallel graphs that represents a "finer" ordering than the given semi-order and that models the class of solutions found by the algorithm. As before, we simplify the treatment by considering a single building block of the semi-order, the hypothesis being implicit that we then apply our considerations sequentially to all blocks. Let us take again the "elementary" building block of a semi-order, as in (2). Let $\preceq$ be the partial order that defines it, that is, let

$$\tilde{\prec} = (S \times T) \cup (T \times U) \cup (S \times V) \cup (R \times U) \cup (R \times V) \tag{26}$$

and let $\prec$ be the transitive closure of $\tilde{\prec}$ and $\preceq$ the reflexive closure of $\prec$. Consider now sets $R' \subseteq R$, $T' \subseteq T$, and $V' \subseteq V$; define

$$\tilde{R} = R - R' \qquad \tilde{T} = T - T' \qquad \tilde{V} = V - V' \tag{27}$$

and the relation

$$\sqsubset_* = (S \cup R') \times (\tilde{R} \cup T') \cup (\tilde{R} \cup T') \times (\tilde{T} \times V') \cup (\tilde{T} \times V') \times (U \cup \tilde{V}) \tag{28}$$

Furthermore, let $\sqsubset$ be the transitive closure of $\sqsubset_*$ and $\sqsubseteq$ the reflexive closure of $\sqsubset$. From the construction, it is easy to check that $\preceq \subseteq \sqsubseteq$. The relation $\sqsubseteq$ induces a weak order in E which can be represented as

$$\begin{array}{ccccccc} \mathtt{S} & & \tilde{\mathtt{R}} & & \tilde{\mathtt{T}} & & \tilde{\mathtt{V}} \\ \mathtt{R}' & \Longrightarrow & \mathtt{T}' & \Longrightarrow & \mathtt{V}' & \Longrightarrow & \mathtt{U} \end{array} \tag{29}$$

Note that the algorithm S-MATCH(n) finds a linearization $\leq^*$ of E such that there are R', T' and V' such with $\sqsubseteq \subseteq \leq^*$, and these R', T', V' are exactly the sets found by the algorithm. Also, there are states s_1, s_2, and s_3 such that

$$\begin{aligned} S \cup R' &\models^{\mathrm{W}} \phi[\rightarrow s_1] \\ \tilde{R} \cup T' &\models^{\mathrm{W}} \phi[s_1 \rightarrow s_2] \\ \tilde{T} \cup V' &\models^{\mathrm{W}} \phi[s_2 \rightarrow s_3] \\ \tilde{V} \cup U &\models^{\mathrm{W}} \phi[s_3 \rightarrow] \end{aligned} \tag{30}$$

In general, the algorithm will produce a weak order $\sqsubseteq$

$$\mathtt{Q}_1 \Longrightarrow \mathtt{Q}_1 \Longrightarrow \cdots \Longrightarrow \mathtt{Q}_q \tag{31}$$

and a collection of states $s_0, \ldots, s_q$, where s_0 is the initial state and s_q a final state such that

$$Q_k \models^{\mathrm{W}} \phi[s_{k-1} \rightarrow s_k]. \tag{32}$$

Lemma 3 *Let $\preceq$ and $\sqsubseteq$ be two partial orders with $\preceq \subseteq \sqsubseteq$, and E a set; if $(E, \sqsubseteq) \models^{\mathrm{W}} \phi$, then $(E, \preceq) \models^{\mathrm{W}} \phi$.*

Proof. If $(E, \sqsubseteq) \models^{\mathrm{W}} \phi$, then there is a total order $\leq$ such that $\sqsubseteq \subseteq \leq$ and $(E, \leq) \models \phi$. But since $\preceq \subseteq \sqsubseteq \subseteq \leq$, $\leq$ is a linearization of $\preceq$ as well, so $(E, \preceq) \models^{\mathrm{W}} \phi$. □

Lemma 4 *Let $(E, \preceq)$ be a semi-order, and ϕ a regular expression. Let $(E, \sqsubseteq)$ be the weak order of type (31) produced by the algorithm on $(E, \preceq)$ with expression ϕ. Then*

$$(E, \sqsubseteq) \models^{\mathrm{W}} \phi \tag{33}$$

and $\preceq \subseteq \sqsubseteq$.

Proof. The algorithm produces $Q_1, \ldots, Q_q$ with $E = Q_1 \cup \cdots \cup Q_q$ such that (32) holds. So, for each multiset $(Q, \emptyset)$ there is a string

$$\omega_i = \omega_{i1} \cdots \omega_{i,k_i} \tag{34}$$

with $Q_i = \{\!\{\omega_{i1}, \ldots, \omega_{i,k_i}\}\!\}$ such that $\omega_i \models^{\mathrm{W}} \phi[s_{k-1} \rightarrow s_k]$. Clearly $\omega = \omega_1 \cdots \omega_q$ is a linearization of $(E, \sqsubseteq)$, and

$$\omega \models^{\mathrm{W}} \phi[s_0 \rightarrow s_1]\phi[s_1 \rightarrow s_2] \cdots \phi[s_{q-1} \rightarrow s_q]. \tag{35}$$

By lemma 2,

$$\omega \models^{\mathrm{W}} \phi[s_0 \to s_q]. \tag{36}$$

Since s_0 is initial and s_q final, $\omega \models^{\mathrm{W}} \phi$. □

Lemma 5 *Suppose* $(E, \preceq) \models^{\mathrm{W}} \phi$*, where* $\preceq$ *is a semi-order. Then the algorithm* S-MATCH *produces a weak order* $(E, \sqsubseteq)$ *such that* $(E, \sqsubseteq) \models^{\mathrm{W}} \phi$ *and* $\leq \subseteq \sqsubseteq$.

Proof. We prove the lemma by induction on the number of layers of the semi-order.

If the semi-order only has one layer, then S-MATCH reduces to MULTI-SET-MATCH(n).

Suppose then that the semi-order has n layers, and let the first part of it be labeled as in (2) (the set T being possibly empty). Since $(E, \preceq) \models^{\mathrm{W}} \phi$ there is a total order $(E, \leq) \models \phi$. This total order places the elements of E in a list $\omega = \omega_1 \cdots \omega_n$ such that $\omega \models \phi$. We now create a weak order $(E, \sqsubseteq)$ as follows.

Set $\omega_s = \max\{\omega_k | \omega_k \in S\}$ (S being part of a layer, as indicated in)2)), where *max* is relative to the total order $\leq$. Let $Q_1 = \{\omega_k | \omega_k \leq \omega_s\}$, then:

i) $S \subseteq Q_1$
ii) $S_1 \subseteq S \cup R$

Property i) is obvious from the definition. For property ii), suppose there is ω_i such that $\omega_i \in Q_1$ and $\omega_i \notin S \cup R$. It is clearly $\omega_i \neq \omega_s$, since $\omega_s \in S$. By the structure of the semi-order, if $\omega_i \notin S \cup R$, then $\omega_s \prec \omega_i$; on the other hand, if $\omega_i \in Q_1$, then $\omega_i \leq \omega_s$, which violates the hypothesis that $\leq \subseteq \preceq$.

Q_1 contains a prefix of the string ω, so there is a state s_1 such that

$$\omega_1 \cdots \omega_s \models \phi[\to s_1] \tag{37}$$

that is

$$Q \models^{\mathrm{W}} \phi[\to s_1] \tag{38}$$

Q_1 is a sub-set of $S \cup R$, and the algorithm tries all possible subsets and truncations of $R \cup S$, so it will find (among possibly others), Q_1 and $\phi[\to s_1]$, set $\tilde{R} = R - R'$ and create a new semi-order in which it will repeat the procedure.

Let W be the semi-order obtained from E by removing the events in Q_1. The algorithm will match W against $\phi[s_1 \to]$. By lemma 2 and (38), we have

$$\begin{array}{lcll} \omega_1 \cdots \omega_n & \models & \phi & \\ & \models & \phi[\to s_1]\phi[s_1 \to] & \text{(lemma 2)} \\ \omega_1 \cdots \omega_s & \models & \phi[\to s_1] & \text{(eq. (37))} \\ \hline \omega_{s+1} \cdots \omega_n & \models & \phi[s_1 \to] & \text{(lemma 2)} \end{array} \tag{39}$$

so that $(W, \preceq) \models^{\mathrm{W}} \phi[s_1 \to]$. Since W has less than n layers, by the inductive hypothesis S-MATCH will succeed and find the solution. □

With these preliminaries, it is now quite easy to prove the main result in this section:

Theorem 3 *Algorithm* S-MATCH *applied to a semi-order* $(E, \preceq)$ *and an expression* ϕ *stops with success if and only if* $(E, \preceq) \models^{\mathrm{W}} \phi$.

Proof. If $(E, \preceq) \models^{\mathrm{W}} \phi$, then lemma 5 shows that the algorithm stops with success.

If the algorithm stops with success then, by lemma 4, there is a weak order $(E, \sqsubseteq)$ such that $(E, \sqsubseteq) \models^{\mathrm{W}} \phi$ and $\preceq \subseteq \sqsubseteq$. By lemma 3, this entails that $(E, \preceq) \models^{\mathrm{W}} \phi$. □

Bibliography

[All83] J. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM* 26(11), 1983.

[GJ79] M. R. Garey, D. S. Johnson. *Computers and Intractability; A guide to the theory of NP-Completeness*. New York:Freeman and Company, 1979.

[KT10] E. Kopczyński, A. W. To. Parikh images of grammars: complexity and applications. In *Proceedings of the 25th annual IEEE symposium on logic in computer science*. Pp. 80–9. 2010.

[Par66] R. J. Parikh. On context-free languages. *Journal of the ACM* 13(4):570–81, 1966.

[PV97] M. Pirlot, P. Vincke. *Semiorders. Properties, representations, applications*. Kluwer, 1997.

[San10] S. Santini. On the semantics of complex events in videos. In *Proceedings of the ACM International Workshop on Events in Multimedia*. ACM, 2010.

[San12] S. Santini. Regular languages with variables on graphs. *Information and Computation* 211:1–28, 2012.

Madrid, April 2012

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): Actas de las "XII Jornadas sobre Programación y Lenguajes (PROLE'2012), Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

String-based Multi-adjoint Lattices for Tracing Fuzzy Logic Computations

Pedro J. Morcillo[1], **Ginés Moreno**[1], **Jaime Penabad**[1] **and Carlos Vázquez**[1]

[1] PedroJ.Morcillo@alu.uclm.es Gines.Moreno@uclm.es
Jaime.Penabad@uclm.es Carlos.Vazquez@alu.uclm.es
Faculty of Computer Science Engineering
University of Castilla-La Mancha
02071 Albacete (Spain)

Abstract: Classically, most programming languages use in a predefined way the notion of "string" as an standard *data structure* for a comfortable management of arbitrary sequences of characters. However, in this paper we assign a different role to this concept: here we are concerned with fuzzy logic programming, a somehow recent paradigm trying to introduce fuzzy logic into logic programming. In this setting, the mathematical concept of *multi-adjoint lattice* has been successfully exploited into the so-called *Multi-adjoint Logic Programming* approach, MALP in brief, for modeling flexible notions of truth-degrees beyond the simpler case of true and false. Our main goal points out not only our formal proof verifying that string-based lattices accomplish with the so-called *multi-adjoint property* (as well as its Cartesian product with similar structures), but also its correspondence with interesting debugging tasks into the FLOPER system (from *"Fuzzy LOgic Programming Environment for Research"*) developed in our research group.

Keywords: Cartesian Product of Multi-adjoint Lattices; Fuzzy (Multi-adjoint) Logic Programming; Declarative Debugging

1 Introduction

In essence, the notion of multi-adjoint lattice considers a *carrier* set L (whose elements verify a concrete ordering $\leq$) equipped with a set of connectives like implications, conjunctions, disjunctions and other *hybrid operators* (not always belonging to an standard taxonomy) with the particularity that for each implication symbol there exists its adjoint conjunction used for modeling the *modus ponens* inference rule in a fuzzy logic setting. For instance, some adjoint pairs -i.e. conjunctors and implications- in the lattice $([0,1],\leq)$ are presented in the following paragraph (from now on, this lattice will be called $\mathcal{V}$ along this paper), where labels `L`, `G` and `P` mean respectively *Łukasiewicz logic*, *Gödel intuitionistic logic* and *product logic* (with different capabilities for modeling *pessimist*, *optimist* and *realistic scenarios*, respectively):

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): PROLE'2012, pp. 177-191, ISBN:978-84-15487-27-2.
Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

$$\begin{array}{llllll}
\&_{\mathtt{P}}(x,y) & \triangleq & x*y & \leftarrow_{\mathtt{P}}(x,y) & \triangleq & \min(1,x/y) \quad \textit{Product} \\
\&_{\mathtt{G}}(x,y) & \triangleq & \min(x,y) & \leftarrow_{\mathtt{G}}(x,y) & \triangleq & \begin{cases} 1 & \text{if } y \leq x \\ x & \text{otherwise} \end{cases} \quad \textit{Gödel} \\
\&_{\mathtt{L}}(x,y) & \triangleq & \max(0,x+y-1) & \leftarrow_{\mathtt{L}}(x,y) & \triangleq & \min\{x-y+1,1\} \quad \textit{Łukasiewicz}
\end{array}$$

Moreover, in the MALP framework [MOV04], each program has its own associated multi-adjoint lattice and each program rule (whose syntax, described in detail in Section 3, extends very significantly a Prolog clause[1]) is "weighted" with an element of L, whereas the components in its body are *linked* with connectives of the lattice. For instance, in the following propositional MALP program (where obviously $@_{\mathtt{aver}}$ refers to the classical average operator):

$$\begin{array}{llll}
p & \leftarrow_{\mathtt{P}} & @_{\mathtt{aver}}(q,r) & \textit{with} \quad 0.9 \\
q & \leftarrow & & \textit{with} \quad 0.8 \\
r & \leftarrow & & \textit{with} \quad 0.6
\end{array}$$

the last two rules directly assign truth values 0.8 and 0.6 to propositional symbols q and r, respectively, and the execution of p using the first rule, simply consists in evaluating the expression "$\&_{\mathtt{P}}(0.9, @_{\mathtt{aver}}(0.8,0.6))$", which returns the final truth degree 0.63.

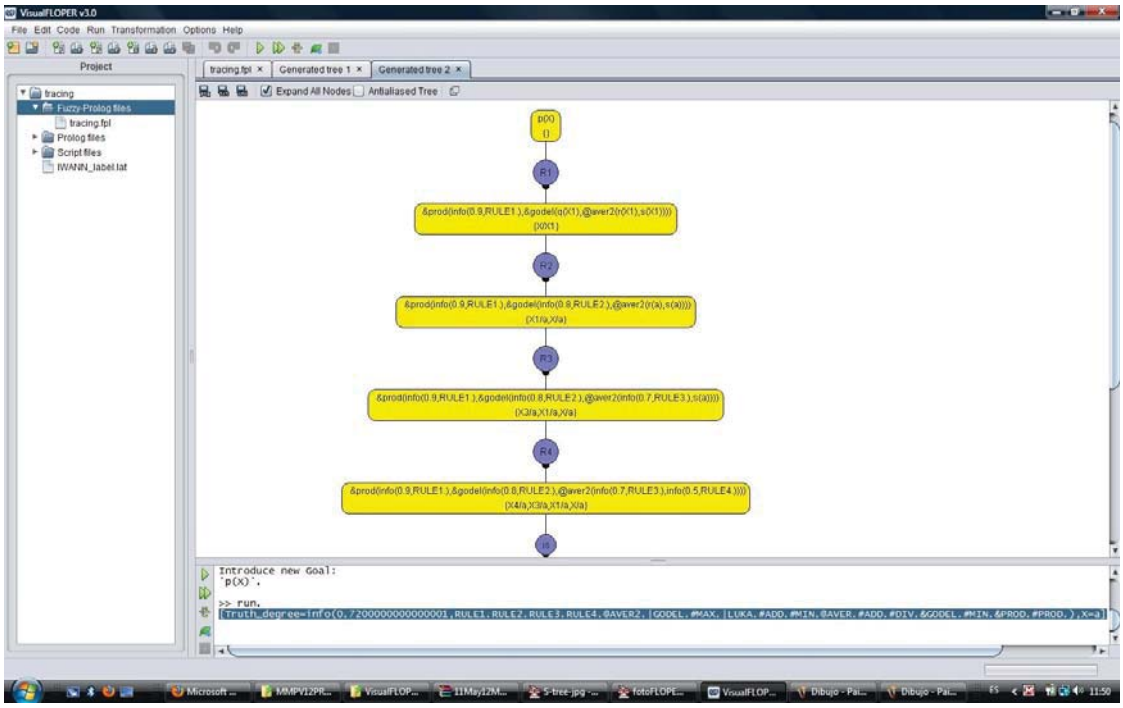
In the following section we describe the shape of the elements, ordering relation and behaviour of the connectives of new multi-adjoint lattices obtained by applying the Cartesian product to previous ones. The main application of such structures into the MALP framework is that it is very easy to attach to program rules and fuzzy connectives "labels" related not only with truth degrees, but also with "augmented information" very useful for designing further *debugging* tasks devoted to "document" proof procedures. Our work is inspired by [RR08, RR09], where authors use the so-called *qualification domain of weights* $\mathscr{W}$ for counting the number of computational steps performed along derivations: however, our proposed technique surpass such approach by providing deeper details on the nature of every fuzzy-logic evaluation step.

Moreover, in Section 3 we present the syntax and procedural semantics of the MALP framework, which exploits multi-adjoint lattices for modeling richer notions of truth degrees to be managed by fuzzy programs. Next, we present the FLOPER tool recently equipped with a graphical interface as shown in Figure 1 [MMPV10, MMPV11c, MMPV11a] (please, visit the web page `http://dectau.uclm.es/floper/`), which currently is successfully used for compiling (to standard Prolog code), executing and debugging MALP programs in a safe way and it is ready for being extended in the near future with powerful transformation and optimization techniques designed in our research group in the recent past [JMP05, GM08].

After describing some guidelines for easily managing multi-adjoint lattices expressed by means of Prolog clauses into the FLOPER system, in Section 4 we propose a sophisticated kind of lattices based on the Cartesian product of previous lattices (based on strings) capable for taking into account details on declarative traces, such as the sequence of computations (regarding program rules, fuzzy connectives and primitive operators) needed for evaluating a given goal. Finally, in Section 5 we give our conclusions and provide some lines for future work.

[1] We assume familiarity with pure Logic Programming [Llo87, JA07] and its most popular language Prolog.

Figure 1: Screen-shot of a work session with FLOPER.



2 String-based Multi-adjoint Lattices and Cartesian Product

In this section we focus on the theoretical results which guarantee that a lattice based on strings is also a multi-adjoint lattice, as well as its Cartesian product with any other multi-adjoint lattice (this kind of sophisticated lattices can be associated to MALP programs in order to report at execution time, a detailed description of the computational steps performed for reaching solutions). We start this section with the formal definition of multi-adjoint lattice.

Definition 1 Let $(L, \leq)$ be a lattice. A *multi-adjoint lattice* is a tuple $(L, \leq, \leftarrow_1, \&_1, \ldots, \leftarrow_n, \&_n)$ such that:

i) $(L, \leq)$ is a complete lattice, namely, $\forall S \subset L, S \neq \emptyset, \exists\, inf(S), sup(S)$[2].

ii) $(\&_i, \leftarrow_i)$ is an *adjoint pair* in $(L, \leq)$, i.e.:

1) $\&_i$ is increasing in both arguments, for all i, $i = 1, \ldots, n$.

2) $\leftarrow_i$ is increasing in the first argument and decreasing in the second, for all i.

3) $x \leq (y \leftarrow_i z)$ if and only if $(x \&_i z) \leq y$, for any $x, y, z \in L$ (adjoint property).

iii) $\top \&_i v = v \&_i \top = v$ for all $v \in L, i = 1, \ldots, n$, where $\top = sup(L)$.

[2] Then, it is a bounded lattice, i.e. it has bottom and top elements, denoted by $\bot$ and $\top$, respectively.

This last condition, called *adjoint property*, is the most important feature of the framework. From now, we are going to focus on the classical notion of Cartesian product applied on these structures, which necessarily returns objects inheriting the required properties of such lattices.

Theorem 1 *If $L_1,\ldots,L_n$ are multi-adjoint lattices, then its Cartesian product $L = L_1 \times \cdots \times L_n$ is also a multi-adjoint lattice.*

Proof. In order to simplify our sketch but without loss of generality, we only consider two multi-adjoint lattices $(L_1,\leq_1,\&_1,\leftarrow_1)$ and $(L_2,\leq_2,\&_2,\leftarrow_2)$, each one equipped with just a single adjoint pair. $L = L_1 \times L_2$ has lattice structure with an order induced in the product from $(L_1,\leq_1)$ and $(L_2,\leq_2)$ as follows: $(x_1,y_1) \leq (x_2,y_2) \Leftrightarrow x_1 \leq_1 x_2,\ y_1 \leq_2 y_2$. Moreover, being $\top_1 = sup(L_1)$, $\bot_1 = inf(L_1)$, $\top_2 = sup(L_2)$ and $\bot_2 = inf(L_2)$, we have that $(\top_1,\top_2) = sup(L)$ and $(\bot_1,\bot_2) = inf(L)$, which implies that the Cartesian product L is a bounded lattice if both L_1 and L_2 are also bounded lattices.

Analogously, $L_1 \times L_2$ is a complete lattice if L_1 and L_2 verify too the same property.

Finally, from the adjoint pairs $(\&_1,\leftarrow_1)$ and $(\&_2,\leftarrow_2)$ in L_1 and L_2, respectively, it is possible to define the following connectives in L: $(x_1,y_1)\&(x_2,y_2) \triangleq (x_1\&_1x_2, y_1\&_2y_2)$ and $(x_1,y_1) \leftarrow (x_2,y_2) \triangleq (x_1 \leftarrow_1 x_2, y_1 \leftarrow_2 y_2)$, for which it is easy to justify that they conform an adjoint pair in $L_1 \times L_2$ (thus satisfying, in particular, the adjoint property).

In a similar way, it is also possible to define new connectives (conjunctions, disjunctions, etc.) in the Cartesian product $L_1 \times L_2$ from the corresponding pairs of operators defined in both lattices L_1 and L_2. □

Moreover, if we are interested in knowing more detailed data about the set of program rules and connective definitions evaluated for obtaining each solution then it will be mandatory to use a new lattice $\mathscr{S}$ based on *strings* or *labels* (i.e., sequences of characters) for generating the Cartesian product $\mathscr{V} \times \mathscr{S}$. In order to achieve our purposes, we firstly must show not only that $\mathscr{S}$ is a multi-adjoint lattice, but also that the concatenation operation of strings, usually called *append* in many programming languages, plays the role of an adjoint conjunction in such lattice (this last condition is required by practical aspects which are explained in Section 4).

When trying to solve both problems, we have analyzed several alternatives for establishing ordering relations among the elements of $\mathscr{S}$, such as the classical lexicographic ordering typically used for sorting words in dictionaries, or those ones based on prefixes, sub-strings, etc., (for instance 'ab' and 'bc' are respectively a prefix and a sub-string of 'abcd'). Unfortunately we have observed that it is never possible to prove that *append* acts as a t-norm.

However, we have recently conceived an alternative, second way for granting that $\mathscr{S}$ is really a multi-adjoint lattice, which definitely solves our problems. The clever point is to "view" each string as a natural number by associating to each character its corresponding ASCII code. So, it is possible to establish a bijective mapping $[\,]$ with $\mathbb{N}$.

Let be $A = \{a_0,\ldots,a_{n-1}\}$ a set, called alphabet, whose elements are symbols. A string s over A is a finite sequence of elements of this set, that is, $s = a_1 \ldots a_m$, where $a_i \in A$, $i = 1,\ldots,m$. The set of all strings over A, denoted by S, is defined as $S = \cup_{k\in\mathbb{N}} A^k$. The definition of S guarantees its numerable character, that is, S contains a (numerable) infinite number of strings like $a_1 \ldots a_m$ formed by elements of A. Although the mentioned numerable character is obtained from well

known results of set theory, in Theorem 2 we will justify it by formalizing a bijection (and its reverse function) from S to $\mathbb{N}$ which will be relevant in the multi-adjoint scope.

Each element of A^k is a string of k elements (with length k) that can be viewed as words; in this case, S could be interpreted as the set of all words with finite length. This interpretation is attractive since each formal language with alphabet A is the set of formulae constructed with elements or words of S that are, also, constrained to the syntactic rules established by that language. Any language with alphabet A would be, accordingly, a subset of S.

Moreover, in set S we define the concatenation, denoted by $\cdot$, as the internal operation

$$A^p \times A^q \to A^{p+q}$$

$$(s,t) \mapsto s \cdot t$$

such that, given $s = a_1 \dots a_p, t = b_1 \dots b_q$, then $s \cdot t = a_1 \dots a_p b_1 \dots b_q$.

Let be the primitive functions $cod : A \to Im(cod)$ and $asc : Im(cod) \to A$, such that $cod(a_i) = i$ and $asc(i) = a_i$. It is trivial to proof that cod is the inverse of function asc and reciprocally. Symbol $''$ represents the empty (with length 0) string. We define the map $[\,] : S \to \mathbb{N}$ through rules R_1 and R_2:

$$\begin{array}{ll} R_1: & [''] = 0 \\ R_2: & [a_1 \dots a_m] = (cod(a_1)+1)n^{m-1} + \dots + (cod(a_m)+1)n^0 \end{array}$$

where $a_i \in A, \forall i$, and $a_1 \dots a_m \in S$. Rule R_2 can be rewritten as:

$$R_2^*: \quad [s.a] = [s]n + cod(a) + 1$$

where $s.a$ indicates a string obtained after adding the symbol a at the end of string s, $s.a = a_1 \dots a_m.a = a_1 \dots a_m a$.

In order to illustrate this mapping, consider the alphabet $\{a,b,c\}$ and the string $s = cab$, where $[cab] = (cod(c)+1)3^2 + (cod(a)+1)3 + (cod(b)+1) = 3*9+1*3+2 = 32$.

Theorem 2 *The map $[\,]$, defined by R_1 and R_2 (or R_2^*), is bijective.*

Proof. Indeed, it is map since each string $s \in S$ is associated with a unique natural number. In order to proof the bijective character of $[\,]$, we will demonstrate that its inverse function is the map $<>: \mathbb{N} \to S$, defined as:

$$\begin{array}{l} <0> = '' \\ <m> = <\lfloor (m-1)/n \rfloor> .asc((m-1)\%n) \end{array}$$

where $\lfloor\,\rfloor : \mathbb{R} \to \mathbb{N}$ is the floor function, so $\lfloor (m-1)/n \rfloor$ is the floor of the quotient, and $(m-1)\%n$ is the remainder modulo n (number of elements of A) of the integer $m-1$.

We will see that compositions $(<> \circ [\,]) : S \to S$ and $([\,] \circ <>) : \mathbb{N} \to \mathbb{N}$ coincide with the identity map (id_S and $id_{\mathbb{N}}$, respectively), which justifies that $[\,]$ is the inverse of $<>$, and reciprocally. Firstly, we prove that $(<> \circ [\,])(s) = s, \forall s \in S$.

1. $(<> \circ [\,])('') = <['']> = <0> = ''$

2. $(<> \circ [\,])(s.a) =$
$< [s.a] >=< [s]n + cod(a) + 1 >=$
$< \lfloor ([s]n + cod(a) + 1 - 1)/n \rfloor > .asc(([s]n + cod(a) + 1 - 1)\%n) =$
$< \lfloor ([s]n + cod(a))/n \rfloor > .asc(([s]n + cod(a))\%n) =$
$< \lfloor [s]n/n \rfloor > .asc(cod(a)\%n) =< \lfloor [s] \rfloor > .asc(cod(a)) =$
$< [s] > .a$

Hence, $< [s.a] >=< [s] > .a$, that is, if $s = a_1 \ldots a_m$, then $< [a_1 \ldots a_m a] >=< [a_1 \ldots a_m] > .a = \cdots =< [''] > .a_1 \ldots . a_m.a = s.a = id_S(s.a)$, where $id_S : S \to S$ is the identity map of S.

Secondly, we will demonstrate that $([]\circ <>)(j) = j, \forall j \in \mathbb{N}$:

1. $([]\circ <>)(0) = [< 0 >] = [''] = 0$

Since j can be expressed as $xn + y$, where $x, y \in \mathbb{N}, 0 < y \leq n$, if $j > 0$, we have

2. $([\,]\circ <>)(j) =$
$([\,]\circ <>)(xn + y) = [< xn + y >] =$
$[< \lfloor (xn + y - 1)/n \rfloor > .asc((xn + y - 1)\%n)] =$
$[< \lfloor (xn + y - 1)/n \rfloor >]n + cod(asc((xn + y - 1)\%n)) + 1 =$
$[< \lfloor xn/n \rfloor >]n + cod(asc(y - 1)) + 1 =$
$[< x >]n + (y - 1) + 1 =$
$[< x >]n + y$

Therefore, $[< xn + y >] = [< x >]n + y, 0 < y \leq n$. That is, j can be expressed as $j = y_1 n^{m-1} + \cdots + y_{m-1} n^1 + y$, with $y_1, \ldots, y_{m-1} \in \{1, \ldots, n\}$, so $[< j >] = [< y_1 n^{m-1} + \cdots + y_{m-1} n^1 + y >] = [< y_1 n^{m-2} + \cdots + y_{m-1} n^0 >]n + y = \cdots = [< 0 >]n + y_1 n^{m-1} + \cdots + y_{m-1} n^1 + y = j$.

Consequently, if $[\,]$ is the inverse function of $<>$ (and reciprocally) both are bijective functions, which proves our desired result. □

In Example 1 we illustrate these maps with a reduced alphabet, while Example 2 works in a real-world setting, giving an idea of the usefulness of this approach.

Example 1 *Consider the alphabet* $A = \{a, b, c\}$. *Then, the string associated to the number* 32 *is* $< 32 >=< \lfloor 31/3 \rfloor > .asc(32\%3) =< 10 > .asc(1) =< \lfloor 9/3 \rfloor > .asc(9\%3).b =< 3 > .asc(0).b =< \lfloor 2/3 \rfloor > .asc(2\%3).a.b =< 0 > .asc(2).a.b =''.c.a.b = cab$

Example 2 *Consider the following strings (using the ASCII code as alphabet):* $s = sea$ *and* $t = son$. *We will apply the* append *operation on them and obtain its associated number.*

- $[s] =$
$[sea] = (cod(s) - 1)128^2 + (cod(e) - 1)128 + (cod(a) - 1) =$
$(115 - 1)128^2 + (101 - 1)128 + (97 - 1) =$
1880672

- $[t] =$
$[son] = (cod(s) - 1)128^2 + (cod(o) - 1)128 + (cod(n) - 1) =$
$(115 - 1)128^2 + (111 - 1)128 + (110 - 1) =$
1881965

- $[s \cdot t] =$
  $[season] = (cod(s)-1)128^5 + (cod(e)-1)128^4 + (cod(a)-1)128^3 + (cod(s)-1)128^2 +$
  $(cod(o)-1)128 + (cod(n)-1) =$
  $(115-1)128^5 + (101-1)128^4 + (97-1)128^3 + (115-1)128^2 + (111-1)128 + 109 =$
  3944056928109

Starting from the reverse usual order $(\mathbb{N}, \leq)$, we define an ordering relation in S that compares strings $s,t \in S$ by

$$s \leq t \iff [s] \leq [t]$$

so the supreme element of S is the empty string, $''$. The application of append over strings *sea* and *son* from the Example 2, which is $s \cdot t = season$, is less than both s and t, that is (by the definition of S), $[s \cdot t] \leq [s]$ and $[s \cdot t] \leq [t]$. As a result of Theorem 2, we have that S is bijective with $\mathbb{N}$. Therefore, the completion (see [MMPV11b]) of S, which is $\mathscr{S} = S \cup \{inf(S)\}$, is bijective with the completion of $\mathbb{N}$, $(\mathbb{N} \cup \{inf(\mathbb{N})\})$, expressed also as $\mathscr{W}$ [RR08, RR09]. So, since $(\mathscr{W}, \leq) = (\mathbb{N} \cup \{inf(\mathbb{N})\}, \leq)$ is a *multi-adjoint lattice*, then $\mathscr{S}$ inherits the same property[3]. After showing that $\mathscr{S}$ is a multi-adjoint lattice via the mapping bijection established with the multi-adjoint lattice $\mathscr{W}$, and before illustrating the benefits that the Cartesian product $\mathscr{V} \times \mathscr{S}$, among others, might play in several software engineering tasks, it is mandatory to explain in the following two sections some details of the MALP language and its associated FLOPER tool.

3 Multi-adjoint Logic Programming and FLOPER

This section summarizes the main features of multi-adjoint logic programming as illustrated in Figure 2 (see [MOV04, JMP09] for a complete formulation of this framework). We work with a first order language, $\mathscr{L}$, containing variables, constants, function symbols, predicate symbols, and several (arbitrary) connectives to increase language expressiveness: implication connectives (denoted by $\leftarrow_1, \leftarrow_2, \ldots$); conjunctive connectives ($\wedge_1, \wedge_2, \ldots$) adjoint conjunctions ($\&_1, \&_2, \ldots$), disjunctive connectives ($\vee_1, \vee_2, \ldots$), and hybrid operators called "aggregators" (usually denoted by $@_1, @_2, \ldots$). Although these connectives are binary operators, we usually generalize them as functions with an arbitrary number of arguments. So, we often write $@(x_1, \ldots, x_n)$ instead of $@(x_1, \ldots, @(x_{n-1}, x_n), \ldots)$. By definition, the truth function for an n-ary connective $[\![@]\!] : L^n \to L$ is required to be monotonous and fulfills $[\![@]\!](\top, \ldots, \top) = \top$, $[\![@]\!](\bot, \ldots, \bot) = \bot$.

Additionally, our language $\mathscr{L}$ contains the values of a multi-adjoint lattice $(L, \leq, \leftarrow_1, \&_1, \ldots, \leftarrow_n, \&_n)$, equipped with a collection of *adjoint pairs* $(\leftarrow_i, \&_i)$ as detailed in previous sections. In general, L may be the carrier of any complete bounded lattice where a L-expression is a well-formed expression composed by values and connectives of L, as well as variable symbols and *primitive operators* (i.e., arithmetic symbols such as $*, +, min$, etc.). In what follows, we assume that the truth function of any connective $@$ in L is given by its corresponding *connective definition*, that is, an equation of the form $@(x_1, \ldots, x_n) \triangleq E$, where E is a L-expression not containing variable symbols apart from $x_1, \ldots, x_n$. See for instance, the classical set of adjoint pairs (conjunctors and implications of *Łukasiewicz logic*, *Gödel intuitionistic logic* and *product logic*) in $([0,1], \leq)$ defined at the beginning of this paper in Section 1.

[3] It is easy to prove that $\&_{\texttt{append}}$ is really an adjoint conjunction in lattice $\mathscr{S}$.

A *rule* is a formula $H \leftarrow_i \mathscr{B}$, where H is an atomic formula (usually called the *head*) and $\mathscr{B}$ (which is called the *body*) is a formula built from atomic formulas $B_1, \ldots, B_n$ — $n \geq 0$ —, truth values of L, conjunctions, disjunctions and other connectives. A *goal* is a body submitted as a query to the system. Roughly speaking, a multi-adjoint logic program is a set of pairs $\langle \mathscr{R}; v \rangle$ (we often write "$\mathscr{R}$ *with* v"), where $\mathscr{R}$ is a rule and v is a *truth degree* (a value of L) expressing the confidence of a programmer in the truth of rule $\mathscr{R}$. By abuse of language, we sometimes refer a tuple $\langle \mathscr{R}; v \rangle$ as a "rule".

The procedural semantics of the multi–adjoint logic language $\mathscr{L}$ can be thought of as an operational phase (based on admissible steps) followed by an interpretive one. In the following, $\mathscr{C}[A]$ denotes a formula where A is a sub-expression which occurs in the –possibly empty– context $\mathscr{C}[]$. Moreover, $\mathscr{C}[A/A']$ means the replacement of A by A' in context $\mathscr{C}[]$, whereas $\mathscr{V}ar(s)$ refers to the set of distinct variables occurring in the syntactic object s, and $\theta[\mathscr{V}ar(s)]$ denotes the substitution obtained from θ by restricting its domain to $\mathscr{V}ar(s)$.

Definition 2 (Admissible Step) Let $\mathscr{Q}$ be a goal and let σ be a substitution. The pair $\langle \mathscr{Q}; \sigma \rangle$ is a *state* and we denote by $\mathscr{E}$ the set of states. Given a program $\mathscr{P}$, an *admissible computation* is formalized as a state transition system, whose transition relation $\rightarrow_{AS} \subseteq (\mathscr{E} \times \mathscr{E})$ is the smallest relation satisfying the following *admissible rules* (where we always consider that A is the selected atom in $\mathscr{Q}$ and $mgu(E)$ denotes the *most general unifier* of an equation set E):

1) $\langle \mathscr{Q}[A]; \sigma \rangle \rightarrow_{AS} \langle (\mathscr{Q}[A/v \,\&_i\, \mathscr{B}])\theta; \sigma\theta \rangle$, if $\theta = mgu(\{H = A\})$ and $\langle H \leftarrow_i \mathscr{B}; v \rangle$ in $\mathscr{P}$.
2) $\langle \mathscr{Q}[A]; \sigma \rangle \rightarrow_{AS} \langle (\mathscr{Q}[A/v])\theta; \sigma\theta \rangle$, if $\theta = mgu(\{H = A\})$ and $\langle H \leftarrow; v \rangle$ in $\mathscr{P}$.

As usual, rules are taken renamed apart. We shall use the symbols $\rightarrow_{AS1}$ and $\rightarrow_{AS2}$ to distinguish between computation steps performed by applying one of the specific admissible rules. Also, the application of a rule on a step will be annotated as a superscript of the $\rightarrow_{AS}$ symbol.

Definition 3 Let $\mathscr{P}$ be a program, $\mathscr{Q}$ a goal and id the empty substitution. An *admissible derivation* is a sequence $\langle \mathscr{Q}; id \rangle \rightarrow_{AS} \cdots \rightarrow_{AS} \langle \mathscr{Q}'; \theta \rangle$. When $\mathscr{Q}'$ is a formula not containing atoms (i.e., a L-expression), the pair $\langle \mathscr{Q}'; \sigma \rangle$, where $\sigma = \theta[\mathscr{V}ar(\mathscr{Q})]$, is called an *admissible computed answer* (a.c.a.) for that derivation.

If we exploit all atoms of a given goal, by applying admissible steps as much as needed during the operational phase, then it becomes a formula with no atoms (a L-expression) which can be then directly interpreted w.r.t. lattice L as follows.

Definition 4 (Interpretive Step) Let $\mathscr{P}$ be a program, $\mathscr{Q}$ a goal and σ a substitution. Assume that $[\![@]\!]$ is the truth function of connective @ in the lattice $(L, \leq)$ associated to $\mathscr{P}$, such that, for values $r_1, \ldots, r_n, r_{n+1} \in L$, we have that $[\![@]\!](r_1, \ldots, r_n) = r_{n+1}$. Then, we formalize the notion of *interpretive computation* as a state transition system, whose transition relation $\rightarrow_{IS} \subseteq (\mathscr{E} \times \mathscr{E})$ is defined as the least one satisfying: $\langle \mathscr{Q}[@(r_1, \ldots, r_n)]; \sigma \rangle \rightarrow_{IS} \langle \mathscr{Q}[@(r_1, \ldots, r_n)/r_{n+1}]; \sigma \rangle$. An *interpretive derivation* is a sequence $\langle \mathscr{Q}; \sigma \rangle \rightarrow_{IS} \cdots \rightarrow_{IS} \langle \mathscr{Q}'; \sigma \rangle$. If $\mathscr{Q}' = r \in L$, being $(L, \leq)$ the lattice associated to $\mathscr{P}$, then $\langle r; \sigma \rangle$ is called a *fuzzy computed answer* (f.c.a.) for that derivation.

From now on, we proceed with more practical aspects regarding multi-adjoint lattices and implementation issues. The parser of our FLOPER tool [MMPV10, MMPV11c] has been imple-

Multi-adjoint logic program:

$$\mathscr{P} = \begin{cases} \mathscr{R}_1: & \langle p(X) & \leftarrow_{\mathtt{P}} & \&_{\mathtt{G}}(q(X), @_{\mathtt{aver}}(r(X), s(X))) & ;0.9\rangle \\ \mathscr{R}_2: & \langle q(a) & \leftarrow & & ;0.8\rangle \\ \mathscr{R}_3: & \langle r(X) & \leftarrow & & ;0.7\rangle \\ \mathscr{R}_4: & \langle s(X) & \leftarrow & & ;0.5\rangle \end{cases}$$

Admissible derivation:

$$\begin{array}{ll} \langle \underline{p(X)};\ id\rangle & \rightarrow_{AS1}{}^{\mathscr{R}_1} \\ \langle \&_{\mathtt{P}}(0.9, \&_{\mathtt{G}}(\underline{q(X_1)}, @_{\mathtt{aver}}(r(X1), s(X1)))); \{X/X_1\}\rangle & \rightarrow_{AS2}{}^{\mathscr{R}_2} \\ \langle \&_{\mathtt{P}}(0.9, \&_{\mathtt{G}}(0.8, @_{\mathtt{aver}}(\underline{r(a)}, s(a)))); \{X/a, X_1/a\}\rangle & \rightarrow_{AS2}{}^{\mathscr{R}_3} \\ \langle \&_{\mathtt{P}}(0.9, \&_{\mathtt{G}}(0.8, @_{\mathtt{aver}}(0.7, \underline{s(a)}))); \{X/a, X_1/a, X_2/a\}\rangle & \rightarrow_{AS2}{}^{\mathscr{R}_4} \\ \langle \&_{\mathtt{P}}(0.9, \&_{\mathtt{G}}(0.8, @_{\mathtt{aver}}(0.7, 0.5))); \{X/a, X_1/a, X_2/a, X_3/a\}\rangle & \end{array}$$

Interpretive derivation:

$$\begin{array}{ll} \langle \&_{\mathtt{P}}(0.9, \&_{\mathtt{G}}(0.8, \underline{@_{\mathtt{aver}}(0.7, 0.5)})); \{X/a\}\rangle & \rightarrow_{IS} \\ \langle \&_{\mathtt{P}}(0.9, \underline{\&_{\mathtt{G}}(0.8, 0.6)}); \{X/a\}\rangle & \rightarrow_{IS} \\ \langle \underline{\&_{\mathtt{P}}(0.9, 0.6)}; \{X/a\}\rangle & \rightarrow_{IS} \\ \langle 0.54; \{X/a\}\rangle & \textbf{\textit{p(a)} has truth degree 0.54} \end{array}$$

Figure 2: MALP program $\mathscr{P}$ with admissible/interpretive derivations for goal $p(X)$.

mented by using the Prolog language. Once the application is loaded inside a Prolog interpreter, it shows a menu which includes options for loading/compiling, parsing, listing and saving MALP programs, as well as for executing/debugging fuzzy goals. Moreover, in [MMPV10] we explain that FLOPER has been recently equipped with new options, called "`lat`" and "`show`", for allowing the possibility of respectively changing and displaying the multi-adjoint lattice associated to a given program.

A very easy way to model truth-degree lattices for being included into the FLOPER tool is also described in [MMPV10], according the following guidelines. All relevant components of each lattice are encapsulated inside a Prolog file which must necessarily contain the definitions of a minimal set of predicates defining the set of valid elements (including special mentions to the "`top`" and "`bottom`" ones), the full or partial ordering established among them, as well as the repertoire of fuzzy connectives which can be used for their subsequent manipulation. In order to simplify our explanation, assume that file "bool.pl" refers to the simplest notion of (a binary) adjoint lattice, thus implementing the following set of predicates:

• `member/1` which is satisfied when being called with a parameter representing a valid truth degree. For instance, in the Boolean case, this predicate can be simply modeled by the Prolog facts `member(0).` and `member(1).`

• `bot/1` and `top/1` obviously answer with the top and bottom element of the lattice, respectively. Both are implemented into "bool.pl" as `bot(0).` and `top(1).`

```
member(X) :- number(X),0=<X,X=<1.                                    bot(0).
leq(X,Y) :- X=<Y.                                                    top(1).

and_luka(X,Y,Z) :- pri_add(X,Y,U1),pri_sub(U1,1,U2),pri_max(0,U2,Z).
and_godel(X,Y,Z):- pri_min(X,Y,Z).
and_prod(X,Y,Z) :- pri_prod(X,Y,Z).

or_luka(X,Y,Z)  :- pri_add(X,Y,U1),pri_min(U1,1,Z).
or_godel(X,Y,Z) :- pri_max(X,Y,Z).
or_prod(X,Y,Z)  :- pri_prod(X,Y,U1),pri_add(X,Y,U2),
                   pri_sub(U2,U1,Z).

agr_aver(X,Y,Z) :- pri_add(X,Y,U),pri_div(U,2,Z).
agr_aver2(X,Y,Z):- or_godel(X,Y,Z1),or_luka(X,Y,Z2),
                   agr_aver(Z1,Z2,Z).

pri_add(X,Y,Z)  :- Z is X+Y.   pri_min(X,Y,Z) :- (X=<Y,Z=X;X>Y,Z=Y).
pri_sub(X,Y,Z)  :- Z is X-Y.   pri_max(X,Y,Z) :- (X=<Y,Z=Y;X>Y,Z=X).
pri_prod(X,Y,Z) :- Z is X * Y. pri_div(X,Y,Z) :- Z is X/Y.
```

Figure 3: Prolog code for representing the multi-adjoint lattice $\mathscr{V}$.

• `leq/2` models the ordering relation among all the possible pairs of truth degrees, and obviously it is only satisfied when it is invoked with two elements verifying that the first parameter is equal or smaller than the second one. So, in our example it suffices with including into "bool.pl" the facts `leq(0,X).` and `leq(X,1).`

• Finally, if we have some fuzzy connectives of the form $\&_{label_1}$ (conjunction), $\vee_{label_2}$ (disjunction) or $@_{label_3}$ (aggregation) with arities n_1, n_2 and n_3 respectively, we must provide clauses defining the *connective predicates* "`and_`*label*$_1$`/(n`$_1$`+1)`", "`or_`*label*$_2$`/(n`$_2$`+1)`" and "`agr_`*label*$_3$`/(n`$_3$`+1)`", where the extra argument of each predicate is intended to contain the result achieved after the evaluation of the proper connective. For instance, in the Boolean case, the following two facts model in a very easy way the behaviour of the classical conjunction operation: `and_bool(0,_,0).  and_bool(1,X,X).`

The reader can easily check that the use of lattice "bool.pl" when working with MALP programs whose rules have the form: "$H \leftarrow_{bool} \&_{bool}(B_1,\ldots,B_n)$ *with* 1", being H and B_i typical atoms, successfully mimics the behaviour of classical Prolog programs where clauses accomplish with the shape "$H :- B_1,\ldots,B_n$". As a novelty in the fuzzy setting, when evaluating goals according to the procedural semantics described in Section 3, each output will contain the corresponding Prolog's substitution (i.e., the *crisp* notion of computed answer obtained by means of classical SLD-resolution) together with the maximum truth degree 1.

As shown in Figure 3, it is also possible to describe by means of Prolog clauses the more flexible lattice $\mathscr{V}$ for working with truth degrees in the infinite space of the real numbers between 0 and 1, allowing too the possibility of using conjunction and disjunction operators recasted from the three typical fuzzy logics proposals described before, as well as two useful descriptions for the hybrid aggregator *average*.

4 Declarative Traces via Cartesian Product of Lattices

As detailed in [MMPV10, MMPV11c], our parser has been implemented by using the classical DCG's (*Definite Clause Grammars*) resource of the Prolog language, since it is a convenient notation for expressing grammar rules. As already commented in the previous section, once the application is loaded inside any Prolog interpreter, it shows a menu which includes options for loading, parsing, listing and saving fuzzy programs, as well as for executing fuzzy goals. All these actions are based in the translation of the fuzzy code into standard Prolog code. The key point is to extend each atom with an extra argument, called *truth variable* of the form "$\_\mathtt{TV_i}$", which is intended to contain the truth degree obtained after the subsequent evaluation of the atom. For instance, the first clause in our target program is translated into:

```
p(X,_TV0) :- q(X,_TV1), r(X,_TV2),s(X,_TV3),agr_aver(_TV2,_TV3,_TV4),
             and_godel(_TV1,_TV4,_TV5),and_prod(0.9,_TV5,_TV0).
```

Moreover, the second clause in our target program in Figure 2, becomes into the pure Prolog fact "`q(a,0.8)`" while a fuzzy goal like "`p(X)`", is translated into the pure Prolog goal: "`p(X, Truth_degree)`" (note that the last truth degree variable is not anonymous now) for which the Prolog interpreter returns the desired fuzzy computed answer $[\mathtt{Truth\_degree} = 0.54, \mathtt{X} = \mathtt{a}]$. The previous set of options suffices for running fuzzy programs (the "`run`" choice also uses the clauses contained in "num.pl", which represent the default lattice) where all internal computations (including compiling and executing) are pure Prolog derivations whereas inputs (fuzzy programs and goals) and outputs (fuzzy computed answers) have always a fuzzy taste, thus producing the illusion on the final user of being working with a purely fuzzy logic programming tool.

```
member(info(X,Y)):-number(X),0=<X,X=<1,atom(Y).   top(info(1,'')).

and_prod(info(X1,X2),info(Y1,Y2),info(Z1,Z2)) :-
            pri_prod(X1,Y1,Z1,DatPROD),pri_app(X2,Y2,Dat1),
            pri_app(Dat1,'&PROD.',Dat2),pri_app(Dat2,DatPROD,Z2).

pri_prod(X,Y,Z,'#PROD.'):-Z is X * Y.

pri_app(X,Y,Z) :-name(X,L1),name(Y,L2),append(L1,L2,L3),name(Z,L3).

append([],X,X).           append([X|Xs],Y,[X|Zs]):-append(Xs,Y,Zs).
.....
```

Figure 4: Lattice modeled in Prolog for representing the Cartesian product $\mathscr{V} \times \mathscr{S}$.

On the other hand, in the previous section we have explained that FLOPER is equipped with two new options, called "`lat`" and "`show`", for allowing the possibility of respectively changing and displaying the multi-adjoint lattice associated to a given program. Assume now that, instead of computing the average of two truth degrees, we prefer to compute the average between the results achieved after applying to both elements the disjunctions operators described by Gödel and Łukasiewicz, that is: $@_{\mathtt{aver_2}}(x_1,x_2) \triangleq @_{\mathtt{aver}}(\vee_{\mathtt{G}}(x_1,x_2), \vee_{\mathtt{L}}(x_1,x_2))$. See the corresponding

Prolog clause modeling such definition listed in Figure 3. Now, by selecting again the "run" option, the system would display the new solution: [Truth_degree = 0.72, X = a].

In what follows, we plan to exploit a much more powerful lattice to cope with more involved *debugging* details based on declarative traces beyond the simpler task described in [RR08, RR09] of counting the number of program rules used during the query-answering procedure (some guidelines can be found in our precedent works [MMPV11c, MMPV11a]). The elements of our intended lattice must contain two components, thus belonging to the Cartesian product $\mathscr{V} \times \mathscr{S}$ seen in Section 2, in order capture not only truth degrees, but also "labels" collecting information about the program rules, fuzzy connectives and primitive operators used for solving goals when executing programs. In order to be loaded into FLOPER, we need to define again the new lattice as a Prolog program, whose elements will be expressed now as data terms of the form "info(Fuzzy_Truth_Degree, Label)" as shown in Figure 4 (we simply show an incomplete but representative set of predicates).

Here, we see that when implementing for instance the conjunction operator of the Product Logic, in the second component of our extended notion of "truth degree", we have *appended* the labels of its arguments with the label '&PROD.' (see clauses defining and_prod, pri_app and append). Of course, in the fuzzy program to be run, we must also take into account the use of labels associated to the program rules. For instance, the set of rules in our example (where we use the complex version of average, i.e., @aver2 in the first rule) must have the form:

```
p(X) <prod &godel(q(X),@aver2(r(X),s(X))) with   info(0.9,'RULE1.').
q(a)                                      with   info(0.8,'RULE2.').
r(X)                                      with   info(0.7,'RULE3.').
s(X)                                      with   info(0.5,'RULE4.').
```

Now the reader can easily check that, after executing goal p(X), we obtain the desired fuzzy computed answer which includes a nice declarative trace collecting the exact sequence of "program-rules, fuzzy-connectives and primitive-operators" evaluated till finding the final solution, which has the following shape by using FLOPER:

```
>> run.
[Truth_degree=info(0.72,  RULE1.RULE2.RULE3.RULE4.
                          @AVER2.|GODEL.#MAX.|LUKA.
                          #ADD.#MIN.@AVER.#ADD.#DIV.
                          &GODEL.#MIN.&PROD.#PROD.),    X=a]
```

In this fuzzy computed answer we obtain both the truth value (i.e., 0.72) and substitution (that is, $X = a$) associated to our goal, but also the sequence of program rules exploited when applying admissible steps (RULE1, RULE2, RULE3 and RULE4, in this order) as well as the list of fuzzy connectives evaluated during the interpretive phase, also detailing the set of primitive operators (of the form #*label*) that they call: in our case, note that we have firstly evaluated aggregator @AVER2 (which calls to connectives |GODEL - defined in terms of the primitive operator #MAX -, |LUKA - which invokes the arithmetic operations #ADD and #MIN - and @AVER - expressed with the use of the primitive operators #ADD and #DIV -), then we need to evaluate the conjunction &GODEL (solved again via the arithmetic symbol #MIN) and the final exploited connective is &PROD (described in terms of the primitive operator #PROD).

$$\langle \underline{p(X)};\ id\rangle \qquad \rightarrow_{AS1}{}^{\mathscr{R}_1}$$

$$\langle \&_{\mathtt{P}}(\mathrm{info}(0.9,\ \texttt{'RULE1.'}), \&_{\mathtt{G}}(\underline{q(X_1)}, @_{\mathtt{aver2}}(r(X1), s(X1)))); \{X/X_1\}\rangle \qquad \rightarrow_{AS2}{}^{\mathscr{R}_2}$$

$$\langle \&_{\mathtt{P}}(\mathrm{info}(0.9,\ \texttt{'RULE1.'}), \&_{\mathtt{G}}(\mathrm{info}(0.8,\ \texttt{'RULE2.'}), @_{\mathtt{aver2}}(\underline{r(a)}, s(a)))); \{X/a\}\rangle \qquad \rightarrow_{AS2}{}^{\mathscr{R}_3}$$

$$\langle \&_{\mathtt{P}}(\mathrm{info}(0.9,\ \texttt{'RULE1.'}), \&_{\mathtt{G}}(\mathrm{info}(0.8,\ \texttt{'RULE2.'}),$$
$$@_{\mathtt{aver2}}(\mathrm{info}(0.7,\ \texttt{'RULE3.'}), \underline{s(a)}))); \{X/a\}\rangle \qquad \rightarrow_{AS2}{}^{\mathscr{R}_4}$$

$$\langle \&_{\mathtt{P}}(\mathrm{info}(0.9,\ \texttt{'RULE1.'}), \&_{\mathtt{G}}(\mathrm{info}(0.8,\ \texttt{'RULE2.'}),$$
$$\underline{@_{\mathtt{aver2}}(\mathrm{info}(0.7,\ \texttt{'RULE3.'}), \mathrm{info}(0.5,\ \texttt{'RULE4.'}))})); \{X/a\}\rangle \qquad \rightarrow_{IS}$$

$$\langle \&_{\mathtt{P}}(\mathrm{info}(0.9,\ \texttt{'RULE1.'}), \underline{\&_{\mathtt{G}}(\mathrm{info}(0.8,\ \texttt{'RULE2.'}), \mathrm{info}(0.85, *))}); \{X/a\}\rangle \qquad \rightarrow_{IS}$$

$$\langle \underline{\&_{\mathtt{P}}(\mathrm{info}(0.9,\ \texttt{'RULE1.'}), \mathrm{info}(0.8, **))}; \{X/a\}\rangle \qquad \rightarrow_{IS}$$

$$\langle \mathrm{info}(0.72,\ \texttt{'RULE1.RULE2.RULE3.RULE4.@AVER2.|GODEL.\#MAX.|LUKA.}$$
$$\texttt{\#ADD.\#MIN.@AVER.\#ADD.\#DIV.\&GODEL.\#MIN.\&PROD.\#PROD.'}); \{X/a\}\rangle.$$

where we have used the following symbols with their corresponding meanings:

* `'RULE3.RULE4.@AVER2.|GODEL.#MAX.|LUKA.#ADD.#MIN.@AVER.#ADD.#DIV.'`

** `'RULE2.RULE3.RULE4.@AVER2.|GODEL.#MAX.|LUKA.#ADD.#MIN.@AVER.#ADD.#DIV.&GODEL.#MIN.'`

Figure 5: Derivation for solving $p(X)$ by using lattice $\mathscr{V} \times \mathscr{S}$.

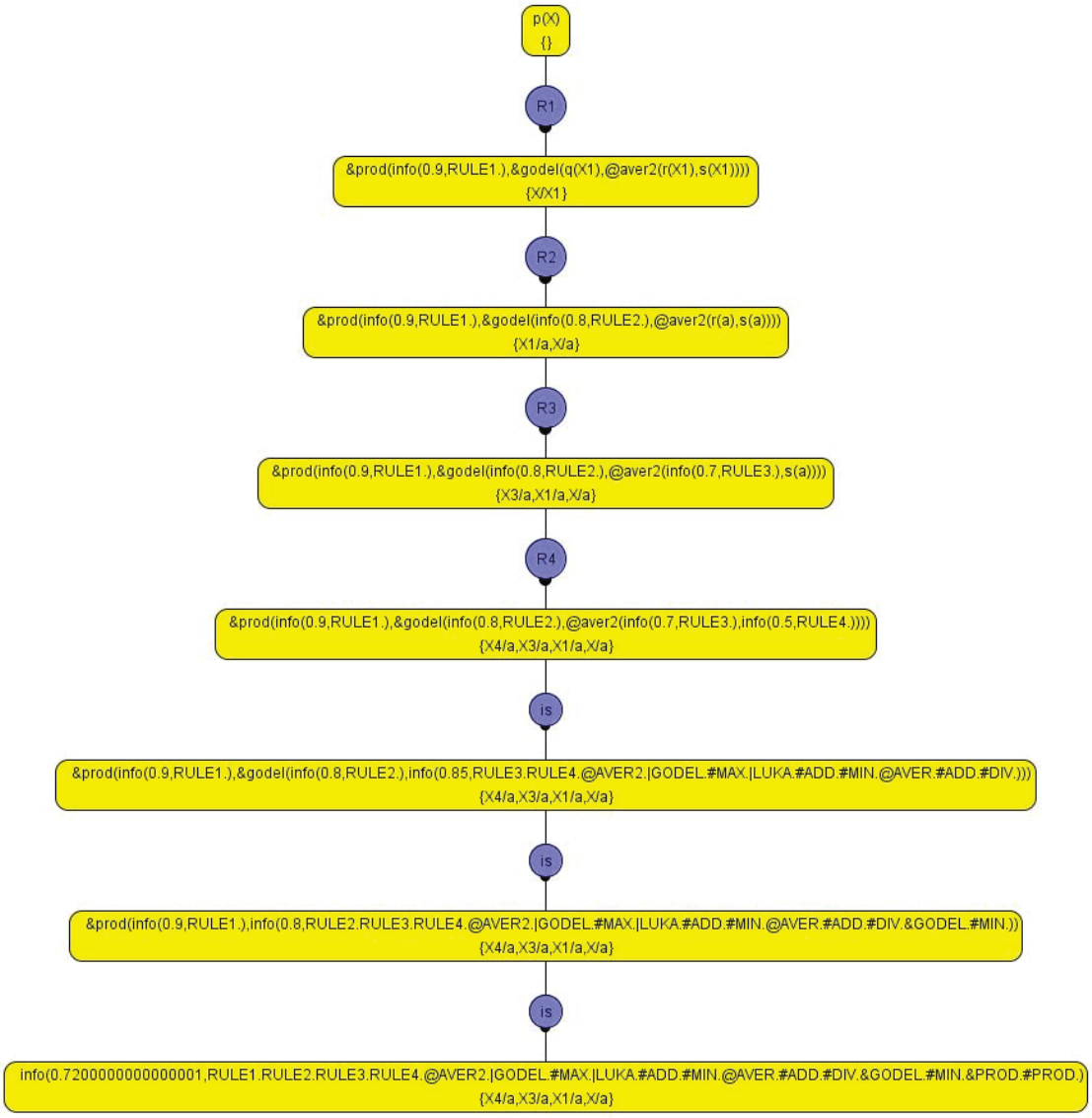
Figures 5 and 6 show the complete derivation built to reach the desired fuzzy computed answer according the procedural semantics described in Section 3.

5 Conclusions and Future Work

This paper has been mainly concerned with theoretical and practical issues focusing into the MALP framework (which could be seen as a very enriched fuzzy extension of Prolog), regarding the use of a string-based multi-adjoint lattice called $\mathscr{S}$. In a more precise way, we have proved and illustrated that the Cartesian product of $\mathscr{S}$ with any other multi-adjoint lattice, is useful to obtain a new, more powerful lattice which can be used for obtaining, at a very low costs, declarative traces on fuzzy computed answers when executing MALP programs inside the FLOPER tool developed in our research group.

We nowadays continue by exploring new uses of such kind of sophisticated lattices in other domains. For instance, we advance in [ALM11, ALM12] that the Cartesian product of $\mathscr{V}$ with a lattice modeling *lists* with a similar shape to $\mathscr{S}$, is very useful for coding with MALP rules a fuzzy variant of the well-known XPath language for the flexible management of XML documents retrieved from the web (our first real-world application using the FLOPER tool can be freely downloaded and tested on-line from `http://dectau.uclm.es/fuzzyXPath/`).

Figure 6: Evaluation tree depicted by FLOPER associated to derivation shown in Figure 5.



Bibliography

[ALM11] J. Almendros-Jiménez, A. Luna, G. Moreno. A Flexible XPath-based Query Language Implemented with Fuzzy Logic Programming. In *Proc. 5th Intl. Symp. on Rules, RuleML'11*. Pp. 186–193. Springer Verlag, Lecture Notes in Computer Science 6826, 2011.

[ALM12] J. M. Almendros-Jiménez, A. Luna, G. Moreno. Fuzzy Logic Programming for Implementing a Flexible XPath-based Query Language. *Electr. Notes Theor. Com-*

put. Sci., Elsevier 282:3–18, 2012.

[GM08] J. Guerrero, G. Moreno. Optimizing Fuzzy Logic Programs by Unfolding, Aggregation and Folding. *Electr. Notes Theor. Comput. Sci., Elsevier* 219:19–34, 2008.

[JA07] P. Julián, M. Alpuente. *Programación Lógica. Teoría y Práctica*. Pearson Educación, S.A., Madrid, 2007.

[JMP05] P. Julián, G. Moreno, J. Penabad. On Fuzzy Unfolding. A Multi-adjoint Approach. *Fuzzy Sets and Systems, Elsevier* 154:16–33, 2005.

[JMP09] P. Julián, G. Moreno, J. Penabad. On the Declarative Semantics of Multi-Adjoint Logic Programs. In *Proc. 10th Intl. Conf. on Artif. Neural Networks, IWANN'09*. Pp. 253–260. Springer, Lecture Notes in Computer Science 5517, 2009.

[Llo87] J. Lloyd. *Foundations of Logic Programming*. Second edition. Springer-Verlag, Berlin, 1987.

[MMPV10] P. Morcillo, G. Moreno, J. Penabad, C. Vázquez. A Practical Management of Fuzzy Truth Degrees using FLOPER. In *Proc. 4th Intl. Symp. on Rule Interchange and Applications, RuleML'10*. Pp. 119–126. Springer Verlag, Lecture Notes in Computer Science 6403, 2010.

[MMPV11a] P. J. Morcillo, G. Moreno, J. Penabad, C. Vázquez. Declarative Traces into Fuzzy Computed Answers. In *Proc. 5th Intl. Symp. on Rules, RuleML'11*. Pp. 170–185. Springer Verlag, Lecture Notes in Computer Science 6826, 2011.

[MMPV11b] P. Morcillo, G. Moreno, J. Penabad, C. Vázquez. Dedekind-Macneille Completion and Multi-Adjoint Lattices. In *Proc. 11th Intl. Conf. on Mathematical Methods in Science and Engineering, CMMSE'11* . Pp. 846–857. Vol. II, 2011.

[MMPV11c] P. Morcillo, G. Moreno, J. Penabad, C. Vázquez. Fuzzy Computed Answers Collecting Proof Information. In *Proc. 11th Intl. Conf. on Artif. Neural Networks, IWANN'11*. Pp. 445–452. Springer Verlag, Lecture Notes in Computer Science 6692, 2011.

[MOV04] J. Medina, M. Ojeda-Aciego, P. Vojtáš. Similarity-based Unification: a multi-adjoint approach. *Fuzzy Sets and Systems* 146:43–62, 2004.

[RR08] M. Rodríguez-Artalejo, C. A. Romero-Díaz. Quantitative logic programming revisited. In *Proc. Functional and Logic Programming, FLOPS'08*. Pp. 272–288. Springer, Lecture Notes in Computer Science 4989, 2008.

[RR09] M. Rodríguez-Artalejo, C. A. Romero-Díaz. Qualified Logic Programming with Bivalued Predicates. *Elec. Notes in Theor. Comp. Sci., Elsevier* 248:67–82, 2009.

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): Actas de las "XII Jornadas sobre Programación y Lenguajes (PROLE'2012), Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

Sesión 6

Ingeniería del Software

Chair: *Dra. Marisa Navarro*

Sesion 6: Ingeniería del Software
Chair: Dra. Marisa Navarro

Dragan Ivanovic, Manuel Carro and Manuel Hermenegildo. *Constraint-Based Runtime Prediction of SLA Violations in Service Orchestrations*.

Jesus M. Almendros-Jimenez and Luis Iribarne. *PTL: A Prolog based Model Transformation Language*.

Enrique Chavarriaga, Fernando Díez and Alfonso Díez. *Intérprete PsiXML para gramáticas de mini-Lenguajes XML en aplicaciones Web*.

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): Actas de las "XII Jornadas sobre Programación y Lenguajes (PROLE'2012), Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

Constraint-Based Runtime Prediction of SLA Violations in Service Orchestrations (extended abstract)

Dragan Ivanović[1], Manuel Carro[2], Manuel Hermenegildo[3]

[1] idragan@clip.dia.fi.upm.es Universidad Politécnica de Madrid
[2] mcarro@fi.upm.es, manuel.carro@imdea.org
[3]herme@fi.upm.es, manuel.hermenegildo@imdea.org
Universidad Politécnica de Madrid and IMDEA Software Institute

Abstract: Quality of Service (QoS) attributes, such as execution time, availability, or cost, are critical for the usability of Web services. This in particular applies to service compositions, which are commonly used for implementing more complex, higher level, and/or cross-organizational tasks by assembling loosely-coupled individual service components (often provided and controlled by third parties). The QoS attributes of service compositions depend on the QoS attributes of the service components, as well as on environmental factors and the actual data being handled, and are usually regulated by means of Service-Level Agreements (SLAs), which define the permissible boundaries for the values of the related properties. Predicting whether an SLA will be violated for a given executing instance of a service composition is therefore very important. Such a prediction can be used for preventing or mitigating the consequences of SLA violations ahead of time.

We propose a method whereby constraints that model SLA conformance and violation are derived at any given point of the execution of a service composition. These constraints are generated using the structure of the composition and properties of the component services, which can be either known or measured empirically. Violation of these constraints means that the corresponding scenario is unfeasible, while satisfaction gives values for the constrained variables (start / end times for activities, or number of loop iterations) which make the scenario possible. These results can be used to perform optimized service matching or trigger preventive adaptation or healing.

The derivation of the constraints that model SLA conformance and violation is based on two key information sources. The first one (called *continuation*) describes the processing that remains to be performed until the end of execution of a given orchestration instance. In general, the continuation is either provided by an orchestration engine, or extracted from its internal state and/or external events. The second information source is the set of assumptions on QoS for the service components used in the orchestration, which are normally empirically collected. The component QoS is described with upper and lower bounds (under some level of confidence), while the prediction is based on (crisp) logical reasoning about the possibility of SLA violation and compliance under the given component bounds.

The constraint-based prediction can be performed at each point of execution for

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): PROLE'2012, pp. 195-196, ISBN:978-84-15487-27-2.
Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

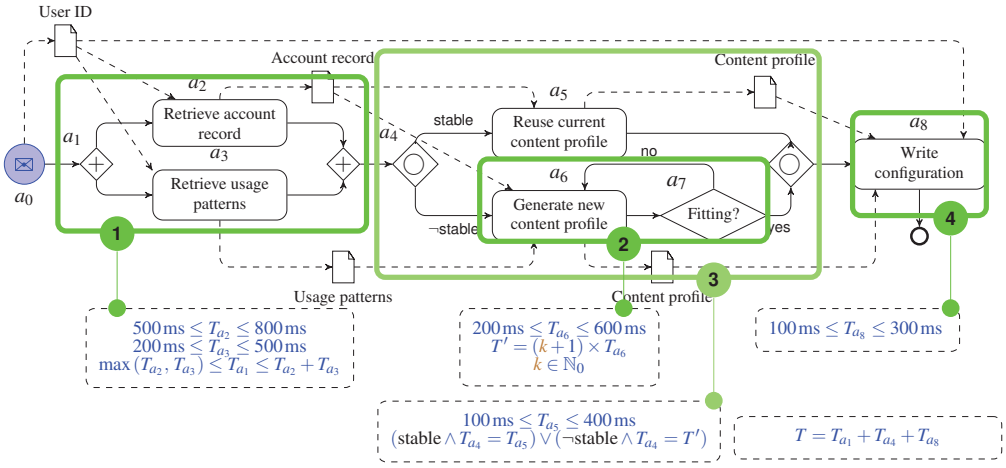


Figure 1: Generation of constraints to predict SLA violations.

which we have the continuation. Unlike data-mining approaches, it does not depend on a historic state of the environment, and can adapt instantly to a dynamic changes in the orchestration code. The first step in prediction is the formulation of the constraint model for the continuation of the running instance (Figure 1), which is dynamically built from the structure of the continuation and the (expected) component QoS bounds. This model expresses the possible range of the remaining QoS for the executing instance (execution time T in case of Figure 1). As an example, the constraints for the structure marked **1** (an and-split) include bounds for the activities and for the structure itself: the components may in fact run in parallel or, if only one thread and one CPU are available, sequentially. The constraints for nested control structures are combined upwards to give the total QoS, T. By constraining T to meet or not the SLA conditions ($T \le T_{\max}$) and $T > T_{\max}$, resp.), we try to rule out one of the two cases, and thus to predict the other (e.g., *SLA compliance ruled out* $\Rightarrow$ *SLA failure predicted*).

Additionally, and since constraints are generated dynamically, we get progressively simpler (and more accurate) systems as the execution proceeds. In addition, we use techniques derived from automatic complexity analysis to derive bounds on the number of loop iterations (k in Figure 1) as functions of the input data, which greatly improves prediction accuracy.

Keywords: Service Orchestrations, Quality of Service, Service Level Agreements, Monitoring, Prediction, Constraints.

A full version of this paper has been published in the Proceedings of ICSOC 2011 [ICH11].

Bibliography

[ICH11] D. Ivanović, M. Carro, M. Hermenegildo. Constraint-Based Runtime Prediction of SLA Violations in Service Orchestrations. In Kappel et al. (eds.), *Service-Oriented Computing – ICSOC 2011*. LNCS 7084, pp. 62–76. Springer Verlag, December 2011. Best paper award.

PTL: A Prolog-based Model Transformation Language

Jesús M. Almendros-Jiménez[1] **and Luis Iribarne**[2]

[1] jalmen@ual.es
[2] luis.iribarne@ual.es
Dpto. de Lenguajes y Computación
Universidad de Almería
04120-Almería (Spain)

Abstract: In this paper we present a model transformation language based on logic programming. The language, called PTL (*Prolog-based Transformation Language*), can be considered as an hybrid language in which ATL-style rules are combined with logic rules for defining transformations. ATL-style rules are used to define mappings from source models to target models while logic rules are used as helpers. The proposal has been implemented so that a Prolog program is automatically obtained from a PTL program. We have equipped our language with debugging and tracing capabilities which help developers to detect programming errors in PTL rules.

Keywords: MDD; Logic Programming; Software Engineering

1 Introduction

Model Driven Engineering (MDE) is an emerging approach for software development. MDE emphasizes the construction of models from which the implementation is derived by applying model transformations. Therefore, *Model Transformation* [Tra05] is a key technology of MDE. According to the *Model Driven Architecture (MDA)* initiative of the *Object Management Group (OMG)* [OMG03], model transformation provides a framework to developers for transforming their models.

MDE proposes (at least) three elements in order to describe a model transformation: the first one is the so-called *meta-meta-model* which is the language for describing meta-models. The second one consists in the *meta-models* of the models to be transformed. Source and target models must conform to the corresponding meta-model. Such meta-models are modeled according to the meta-meta-model. The third one consists in the source and target models. Source and target models are instances of the corresponding meta-models. Furthermore, source and target meta-models are instances of the meta-meta-model. In order to define a model transformation the source and target models are modeled with respect to the meta-meta-model, and source and target meta-models are mapped.

In this context, model transformation needs formal techniques for specifying the transformation. In most of the cases transformations can be expressed with some kinds of *rules*. The rules have to express how source models can be transformed into another. Several transformation languages and tools have been proposed in the literature (see [CH06] for a survey). The most relevant one is the language *ATL (Atlas Transformation Language)* [JABK08] a *domain-specific language* for specifying model-to-model transformations. ATL is a hybrid language, and pro-

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): PROLE'2012, pp. 197-211, ISBN:978-84-15487-27-2.
Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

vides a mixture of declarative and imperative constructs. The declarative part of ATL is based on rules. Such rules define a source pattern matched over source models and a target pattern that creates target models for each match. ATL transformations are unidirectional, operating on read-only source models and producing write-only target models.

In this paper we present a model transformation language based on logic programming. The language, called PTL (*Prolog-based Transformation Language*), can be considered as an hybrid language in which ATL-style rules are combined with logic rules for defining transformations. ATL-style rules are used to define mappings from source models to target models while logic rules are used as helpers. The proposal has been implemented so that a Prolog program is automatically obtained from a PTL program. Hence, PTL makes use of Prolog as transformation engine. The encoding of PTL programs by Prolog is based on a Prolog library for handling meta-models. We have equipped our language with debugging and tracing capabilities which help developers to detect programming errors in PTL rules. Debugging detects PTL rules that cannot be applied to source models, and tracing shows rules and source elements used to obtain a given target model element.

The aim of our work is to provide a framework for model transformation based on logic programming. From a practical point of view, our language can be exploited by Prolog programmers for defining model transformations. So, our proposal aims to introduce model transformation in the declarative paradigm, and can be intended as an application of logic programming (particularly, Prolog) to a context in which rule-based systems are required. Prolog programmers might be interested in programming model transformations; and our language provides the elements involved in model transformation: meta-models handling and mapping rules. We believe that our contribution can be interesting to the logic programming community.

Mapping rules are defined in ATL style syntax. Due to wide acceptance of ATL as transformation language, the adoption of a syntax inspired in ATL to write model transformations makes our approach easier to use and closer to other proposals of transformation languages. The adoption of ATL style syntax facilitates the definition of mappings: basically, objects are mapped to objects of given models. Nevertheless, we retain logic programming as basis, for instance, by defining helpers with Prolog code instead of OCL code, adopted in ATL. It makes that our framework is a mixture of model transformation and logic languages. One of the main ends of helpers in ATL is to serve as query language against the source models. In a logic programming based approach, logic (i.e., Prolog style) rules serve as query language.

Our approach is equipped with debugging and tracing tools. Using Prolog as transformation engine makes possible to program debugging and tracing tools with little effort. Debugging is useful when the target model is *incomplete* while tracing is useful when the target model is *incorrect*.

In summary, our proposal is based on the following elements:

(a) **Encoding to Prolog rules**. ATL-style mapping rules are translated into Prolog rules. Each rule is encoded by a set of Prolog rules, one rule for each object that is created, and one rule for each role ends set by the rule.

(b) **Prolog library for handling meta-models**. Meta-models can be defined in our approach, and a Prolog library for handling meta-models is automatically generated from meta-model specification.

(c) **MOF meta-metamodel**. MOF in our approach is considered as meta-meta-model, and source and target meta-models have to be defined in terms of the MOF meta-model. With this aim, our language can be also used for defining transformations from (and to) the MOF meta-model to (and from) source (and target) meta-models.

(d) **XMI support**. In order to execute a transformation, source models have to be stored in XMI format. Source models are loaded from XMI files, and target models are obtained in XMI format. Our approach allows to define the location of source models and destination of target models. So, our approach can be integrated with XMI-compliant tools.

(e) **Prolog helpers**. Helpers are defined with Prolog rules. Such Prolog rules make use of the Prolog library for handling meta-models.

(f) **Debugging capabilities**. Debugging permits to detect rules that cannot be applied in a certain transformation, and in addition, debugging is able to locate the point of the program in which a certain programming error is found. Such programming error comes from (f.1) Boolean conditions that cannot be satisfied in mapping rules, and (f.2) objects of the target model that cannot be created. When a certain rule cannot be applied, a certain target model element will be missing, and therefore the transformation can be considered erroneous. The debugger is able to give (f.1) the name of rule and the Boolean condition that is false for all the elements of the source model, and (f.2) the name of the rule in which the target model cannot be created, and the failed binding.

(g) **Tracing capabilities**. Tracing permits to visualize how a certain target model element is obtained. Tracing shows all the rules and source model elements that contribute to a target model element. Tracing is useful when a target model element is obtained but is wrong.

Our approach will be applied to a well-known example of model transformation in which a class diagram describing an entity-relationship modeling of a database is transformed into a class diagram representing a relational database. The Prolog-based compiler and interpreter of our PTL language have been developed under *SWI-Prolog*. The code of the case study together with the source code of PTL can be downloaded from http://indalog.ual.es/mdd.

The structure of the paper is as follows. Section 2 will introduce the model transformation setting. Section 3 will present the PTL language. Section 4 will describe the encoding of PTL into Prolog. Section 5 will show how to debug and trace a given PTL transformation. Section 6 will review related work. Finally, Section 7 will conclude and present future work.

2 Model Transformation

In our framework, meta-meta-models are models similar to the model of Figure 1. The meta-meta-model represents the elements of the (UML) class diagram. For instance, a *class* has a *name* and can include attributes, represented by the role *ownedAttribute*, which belongs to the class *Property*. Properties can also be *roles* of *associations*. The members and owned ends of an association are represented by the roles *memberEnd* and *ownedEnd*. *NavigableOwnedEnd* sets navigable role ends. Each property is described with a *name*, a *type*, whether it is *composite* and

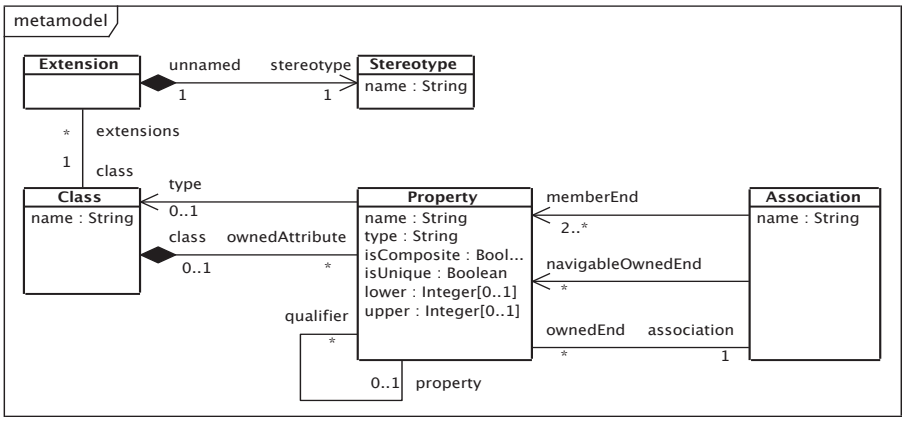


Figure 1: MOF Metamodel of the Class Diagram

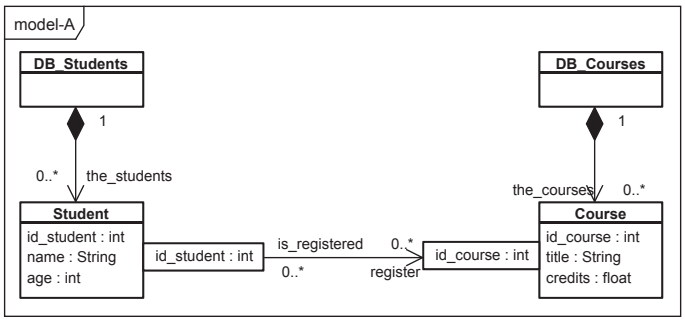


Figure 2: Entity-relationship modeling of the Case Study

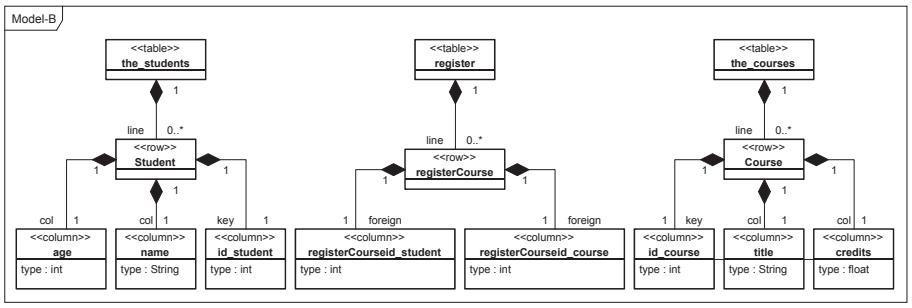


Figure 3: Relational modeling of the Case Study

unique or not, and the *lower* and *upper* bounds of the number of elements of the properties. The association with role *qualifier* links roles to qualifiers of the association. A class can be linked to *extension*'s which are *Stereotype*'s.

In order to define a transformation we have to define the source and the target meta-models. Such meta-models are defined in terms of the MOF meta-meta-model. Source and target models are instances of the corresponding meta-models. Moreover, source and target meta-models are instances of the MOF meta-meta-model.

Now, we would like to show an example of transformation with respect to the meta-meta-model of Figure 1. The model of Figure 2 represents the modeling of a database. We will call this kind of modeling as *"entity-relationship"* modeling of a database in contrast to the model of Figure 3 which will be called *"relational"* modeling of a database.

The entity-relationship modeling of Figure 2 can be summarized as follows. *Entities* are represented by classes (i.e., *Student* and *Course*), including attributes; *Containers* are defined for each entity (i.e., *DB_Students* and *DB_Courses*); *Relationships* are represented by associations. Relation names are association names. Besides, association ends are defined (i.e., *the_students*, *the_courses*, *is_registered* and *register*). Relationships can be adorned with qualifiers and navigability. The role of qualifiers is to specify the key attributes of each entity being foreign keys of the corresponding association.

Figure 3 shows the relational modeling of the same database. Such modeling also defines a class diagram for database design. Tables are composed of rows, and rows are composed of columns. It introduces the following new stereotypes: $<<$ *table* $>>$, $<<$ *row* $>>$ and $<<$ *column* $>>$. Furthermore, *line* is the role of the rows in the table, *key* is the role of the key attributes in rows, *foreign* is the role of the foreign keys in rows, and *col* is the role of non keys and non foreign keys in rows. Finally, each column has an attribute called *type*.

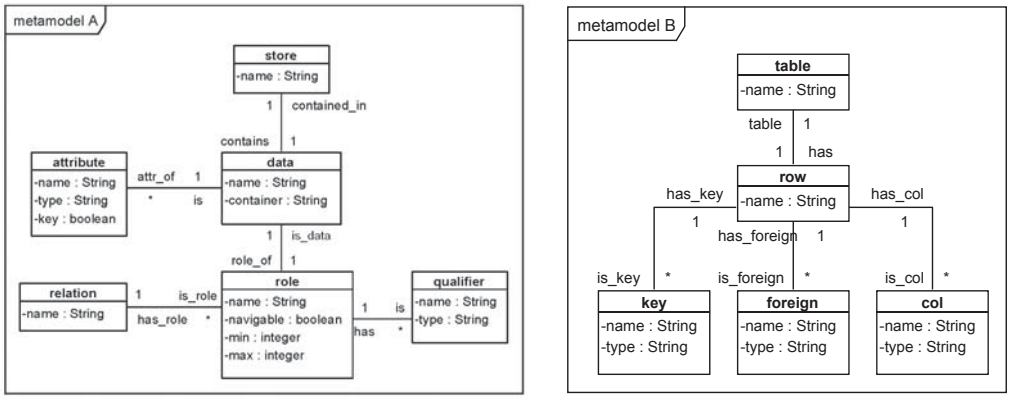


Figure 4: Meta-model of the Source/Target Models

Figure 4 represents the meta-models of both types of modeling. In the first case, *DB_Students* and *DB_Courses* are instances of the class *store*, while *Student* and *Course* are instances of the class *data*, and the attributes of *Student* class and *Course* class are instances of the class *attribute*. In the second case, tables and rows of the target model are instances of the corresponding classes, and the same can be said about *key*, *col* and *foreign* classes.

Now, the problem of model transformation is how to transform a class diagram of the type A (like Figure 2) into a class diagram of type B (like Figure 3). The transformation is as follows.

The transformation generates two tables called *the_students* and *the_courses* each including three columns that are grouped into rows. The table *the_students* includes for each student the attributes of *Student* of Figure 2. The same can be said for the table *the_courses*. Given that the association between *Student* and *Course* is navigable from *Student* to *Course*, a table of

```
metamodel(
er,
[
         class(data, [name,container]),
         class(store, [name]),
         class(attribute, [name,type,key]),
         class(relation, [name]),
         class(role, [name,navigable,min,max]),
         class(qualifier, [name,type]),
         role(contains,store,data,"1","1"),
         role(contained_in,data,store,"1","1"),
         role(attr_of,data,attribute,"0","*"),
         role(is,attribute,data,"1","1"),
         role(has_role,role,relation,"1","1"),
         role(is_role,relation,role,"1","*"),
         role(has,qualifier,role,"1","1"),
         role(is,role,qualifier,"0","*"),
         role(is_data,role,data,"1","1"),
         role(role_of,data,role,"1","1")
]
).
```

Figure 5: Source Metamodel

```
role_id(er, A) :-role(er, A, [name(_), navigable(_), min(_), max(_)]).
data_container(er, A, B) :-data(er, A, [name(_), container(B)]).
attribute_type(er, A, B) :-attribute(er, A, [name(_), type(B), key(_)]).
qualifier_has(er, A, B) :-associationEnds(er, has, A, B).
relation_is_role(er, A, B) :-associationEnds(er, is_role, A, B).
```

Figure 6: Prolog library

pairs is generated to represent the assignments of students to courses, using the role name of the association end, that is, *register* concatenated with *Course*, for naming the cited table. The columns *id_student* and *id_course* taken from qualifiers, play the role of foreign keys which are represented by role *foreign* in the associations of Figure 3.

3 PTL

PTL is a transformation language based on logic programming. A PTL program consists of **meta-model definitions** (source and target meta-models), **mapping rules** and **helpers**.

3.1 Metamodel definitions

Basically, meta-model definitions define meta-model elements: class and roles together with attributes for classes, and for each role the multiplicity and the role ends. From the meta-model definitions the PTL compiler automatically generates a Prolog library for handling the meta-models. Basically, the library contains predicates for each class and role of each meta-model in order to access to class attributes and role ends.

For instance, in Figure 5 we can see the definition of the source meta-model of Figure 4, and the Prolog library of Figure 6 contains (some of) the predicates generated to handle the

rule	:= **rule** rule_name **from** *pointers* [**where** *condition*] **to** *objects*
pointers	:= *pointer* | '(' *pointer*,...,*pointer* ')'
condition	:= *bcondition* | *condition* **and** *condition*
bcondition	:= *access* **==** *access* | *access* **=\=** *access*
pointer	:= pointer_name **:** metamodel **!** class
objects	:= *object*,...,*object*
object	:= *pointer* '(' *binding*,...,*binding* ')'
access	:= value | pointer_name | pointer_name@attribute | pointer_name@role@attribute | helper '(' *access*,...,*access* ')' | **resolveTemp**'(' '(' *access*,...,*access* ')', pointer_name')' | **sequence**'('['*access*,...,*access*']')'
binding	:= attribute <– *access* | role <– *access*

Figure 7: Mapping rules of PTL

er metamodel of Figure 5. We have three kinds of predicates: (a) those for *accessing class attributes*, for instance, *data_container* and *attribute_type*, (b) those for *accessing role ends*, for instance, *qualifier_has* and *relation_is_role*, and (c) a special kind of predicates that retrieve the *identifier* of a certain object (for instance, *role_id*).

The first and third kind of predicates call predicates representing class objects, which are called as the class name, and they have as arguments: the name of the meta-model, the object identifier and a Prolog list with the attributes: each attribute is represented by a Prolog term of the form: *attribute_name(value)*. The second kind of predicates calls to a predicate called *associationEnds*, representing role end objects. The *associationEnds* element has as arguments: the name of the meta-model, the name of the role, and the identifiers of the role ends.

3.2 Mapping rules

The syntax of the mapping rules is described in Figure 7, where *rule_name*, *pointer_name* and *helper* are rule, object and helper names, respectively, and *value* can be any basic datatype.

Basically, a mapping rule maps a set of objects of the source model into a set of objects of the target model. The rule can be conditioned by means of a Boolean condition, including equality (i.e. ==) and dis-equality (i.e. =\=), and the *and* operator. The rule condition is marked with *where*. Objects of target models are defined assigning values to attributes and roles, and they can make use in their definition of attribute values of the source models, *resolveTemp*, helpers and the *sequence* construction. The *resolveTemp* function permits to assign objects created from some other rules. With this aim, *resolveTemp* has as parameters the objects to which the rule is applied, and the name of the object created by the rule.

In Figure 8, we can see also examples of mapping rules in our approach. The rule *table2_er2rl* defines the tables and rows obtained from navigable roles (in the case study, *register* and *registerCourse*). The name of the table is the name of the role, and the name of the row is built from the concatenation of the name of the role, and the name of the role end. Moreover, we have to set the role ends from (to) tables to (from) rows (i.e., *has* and *table*) together with the *is_foreign* role end. The *is_foreign* role end is a sequence of two elements (*registerCourseid_student* and *registerCourseid_course*). For this reason the *sequence* constructor is chosen. Besides, *resolveTemp* retrieves *registerCourseid_course*, and a helper called *inverse_qualifier* is used for the retrieval of *registerCourseid_student*. The rule *foreign2_er2rl* computes the foreign class *regis-*

```
rule table2_er2rl from
p:er!role where (p@navigable==true and p&max=="*") to
     (t:rl!table(
        name <- p@name,
        has <- r
                ),
       r:rl!row(
         name <- concat(p@name,p@is_data@name),
         table <-t,
         is_foreign <- sequence([resolveTemp((p@is,p),f1),inverse_qualifier(p)])
               )
      ).
rule foreign2_er2rl from
(p:er!qualifier,q:er!role) where (p@has == q and q@navigable==false) to
        (f2:rl!foreign(
             name <-concat(concat(q@name,q@is_data@name),p@name),
             type <- p@type,
             has_foreign <- inverse_row(p)
                      )
         ).
```

Figure 8: Examples of PTL Mapping Rules

```
inverse_row(A, E) :-
        associationEnds(er, has, A, B),
        associationEnds(er, has_role, B, C),
        associationEnds(er, is_role, C, D),
        role_navigable(er, D, true),
        resolveTemp(D, r, E).
```

Figure 9: Helpers

terCourseid_student, which is computed from roles and qualifiers which are not navigable. The rule sets the role end *has_foreign* with a helper called *inverse_row*, which computes the row *registerCourse*.

3.3 Helpers

Helpers can de defined with logic rules. For instance, in Figure 9 we can see the definition of the *inverse_row* helper. Helpers can make use of the meta-model Prolog library and the *resolveTemp* construction. It is worth observing that helpers are defined with the following convention: we can define helpers with several arguments, but the last one has to be the result of the helper. In other words, predicates associated to helpers work as functions. The previous convention requires that helpers in PTL mapping rules have n arguments while code of helpers has $n+1$ arguments.

4 Encoding with Prolog rules

Now, we would like to show how PTL mapping rules are encoded with Prolog rules. The schema of the encoding is as follows. Given a PTL mapping rule of the form:

rule *rule_name* **from** *pointers* **where** *bcondition* **to** *objects*

```
(1) object(rl, foreign, L, (A, B), [name(K), type(C)], f2) :-
        foreign2_er2rl((A, B)),
        qualifier_type(er, A, C),
        qualifier_name(er, A, I),
        role_is_data(er, B, E),
        data_name(er, E, G),
        role_name(er, B, F),
        concat(F, G, H),
        concat(H, I, K),
        generate_ids((A, B), f2, L).

(2) associationObjects(rl, has_foreign, C, B) :-
        foreign2_er2rl((A, D)),
        qualifier_id(er, A),
        inverse_row(A, B),
        object(rl, foreign, C, (A, D), _, f2).

(3) foreign2_er2rl((A, B)) :-
        qualifier_id(er, A),
        role_id(er, B),
        qualifier_has(er, A, B),
        role_navigable(er, B, false).
```

Figure 10: Encoded Rule

where $objects \equiv object_1, \ldots, object_n$, $object_i \equiv pointer_i(binding_1, \ldots, binding_k)$ and $pointer_i \equiv q_i : mm_i!class_i$ then the encoding is as follows:

(1) $object(mm_i, class_i, Var, \overline{Vars}, enc[atts], q_i) : -rule\_name(\overline{Vars}'), enc[bind\_att]$.
(2) $associationObjects(mm_i, role_j, Var_1, Var_2) : -rule\_name(\overline{Vars}), enc[role_j <- access]$.
(3) $rule\_name(\overline{Vars}) : -enc[bcondition]$.

where $bind\_att$ is the subset of $binding_1, \ldots, binding_k$ of bindings of the form $attribute <- access$; expressions $role_j <- access$ are each one of the remaining bindings ($j \in \{1, ..., k\}$); and finally, the element $enc[atts]$ is the encoding of attributes, and the elements $enc[bind\_att]$, $enc[role_j <- access]$ and $enc[bcondition]$ are the encoding of such expressions using the Prolog library for meta-models and Prolog helpers.

The predicate *object* encodes the creation of objects of the target model. The *associationObjects* predicate encodes the creation of links between objects of the target model. There are *object* and *associationObjects* rules for each object and each link created by one rule. Hence, one PTL mapping rule is encoded by a set of Prolog rules, one rule for each object that is created, and one rule for each role ends set by the rule. The *object* predicate generates a unique identifier for each object of the target model. With this aim, a Prolog predicate called *generate_ids* is used.

For instance, we can see in Figure 10 how this Prolog library permits to encode the PTL rule *foreign2_er2rl* of Figure 8. The main predicates of the encoding are the predicates *object* (see (1) of Figure 10) and *associationObjects* (see (2) of Figure 10) which define new class objects and role end objects in the target model. They make use of a predicate called the same as the rule, in this case, *foreign2_er2rl* (see (3) of Figure 10). Such predicate retrieves the objects of the source model encoding the Boolean condition of the rule.

Finally, *resolveTemp* construction of PTL can be easily defined with our proposal of encoding:

```
resolveTemp(B, C, A) :- object(_, _, A, B, _, C).
```

```
ptl(Program):-[Program],
                generate_metamodels,
                generate_rules,
                load_models,
                clean_transformation.
```

Figure 11: PTL predicate

```
load_model(A) :-object(A, B, C, D, E, F),
        assert(objectM(A, B, C, D, E, F)),
        fail.
load_model(A) :-associationObjects(A, B, C, D),
        assert(associationObjectsM(A, B, C, D)),
        fail.
load_model(_).
```

Figure 12: Load_model predicate

4.1 PTL interpreter

Now, we would like to show how a PTL program is executed from Prolog. The predicate *ptl* of Figure 11 is called with the file name in which the PTL code is included. For instance:

```
?- ptl('er2rl.ptl').
```

The *ptl* predicate automatically generates the Prolog library of the meta-models defined in the PTL program (predicate *generate_metamodels*), it encodes PTL rules with Prolog rules (predicate *generate_rules*), and it generates the target models from the source models (predicate *load_models*). Since the execution is carried out in main memory, it cleans memory at the end (predicate *clean_transformation*).

The PTL program has to include meta-model definitions (i.e., source and target models), and the set of PTL (mapping and helper) rules. Moreover, we have to specify in the PTL program the location of source and target models with the directives *input* and *output*. These directives also admit to specify the XMI files in which models are stored.

Now, we would like to explain how PTL rules are executed. The previous *load_models* predicate calls to an auxiliary *load_model* predicate, which at the same time, calls to *object* and *associationObjects* predicates, which encode elements created by the rules, and each element is asserted into Prolog (main) memory, in the form of a Prolog fact, called *objectM* and *associationObjectsM*, respectively. The code of *load_model* predicate is shown in Figure 12.

4.2 Transformation execution

In order to execute a particular transformation, we have to specify how source and target meta-models are defined in terms of the MOF meta-meta-model.

With this aim, we have to define a transformation from the MOF meta-meta-model to the source meta-models, and from the target meta-model to the MOF meta-meta-model. It is required in order to load an XMI file containing the source models and to write the target-models into an XMI file[1]. In other words, in order to execute a transformation we have to consider (at least)

[1] A Prolog program has been implemented in order to serialize the MOF meta-model to XMI.

```
transform(Files):-clean_ptl,
                  transform_files(Files).
transform_files([]):-!.
transform_files([F|RF]):-ptl(F),
                  transform_files(RF).
```

Figure 13: Transform predicate

three PTL programs. Hence a transformation is executed as follows:

```
?- transform(['mm2er.ptl','er2rl.ptl','rl2mm.ptl']).
```

The code of the *transform* predicate can be seen in Figure 13.

5 Debugging and Tracing in PTL

In this section, we would like to show how to debug and trace executions in our proposal. Debugging and tracing permit to find programming errors.

5.1 Debugging

Debugging is able to find rules that cannot be applied, and provides the location in which the error is found. PTL mapping rules are not applied due to Boolean conditions that are not satisfied and objects that cannot be created. A Boolean condition is not satisfied whenever a certain equality or inequality is false.

Let us suppose that the PTL rule *table2_er2rl* of Figure 8 includes *p@name*=="*" instead of *p@max*=="*". This is a typical programming error and it cannot be detected by the compiler (i.e, the PTL program is well-typed and syntactically correct). Now, the engineer finds that the target model is wrong. In such a case (s)he can query the debugger, obtaining:

```
?- debugging(['mm2er.ptl','er2rl.ptl','rl2mm.ptl']).
Debugger: Rule Condition of: table2_er2rl cannot be satisfied.
Found error in: role_name
```

The debugger shows the name of the PTL rule (i.e. *table2_er2rl*) that cannot be applied and the found error (i.e. *role_name*).

But the debugger can also detect that target objects cannot be created, for instance, let us suppose that the engineer writes *p@navigable==false*, instead of *p@navigable==true* then the debugger answers as follows:

```
?-  debugging(['mm2er.ptl','er2rl.ptl','rl2mm.ptl']).
Debugger: Objects of: table2_er2rl cannot be created.
Found error in: resolveTemp
```

In this case the objects of rule *table2_er2rl* cannot be created, and the programming error comes from *resolveTemp*, because the call does not succeed.

In summary, debugging is useful when some elements of the target model are not created. In other words, when a certain element of the target model is missing. The PTL interpreter has been modified in order to cover with debugging. The debugger checks PTL rules that cannot be applied and prints the location in which Boolean conditions and object creations fail.

5.2 Tracing

However, the engineer can find a programming error due to the opposite case: a certain target model element is created but it is wrong. In such a case, the engineer can trace from the wrong target element to find the reason (i.e., applied rules and source model elements) of the creation of such element. A target element can be created from a missing Boolean condition.

Let us suppose that in rule *foreign2_er2rl* of Figure 8 the engineer omits the Boolean condition *q@navigable==false*. The engineer reviews the target model and finds a wrong element: *is_registeredStudentid_student*. Now, (s)he can trace the execution from the XMI identifier of the wrong element, obtaining the following answer:

```
?- tracing(['mm2er.ptl','er2rl.ptl',
    'rl2mm.ptl'],'275284307q275325284r2f2c').
Tracing the element: 275284307q275325284r2f2c

Rule: foreign1_rl2mm
Element: foreign
Metamodel: rl

Rule: foreign2_er2rl
Element: qualifier
Metamodel: er

Rule: qualifier_mm2er
Element: association
Metamodel: mm
xmi:id is wfP6O_iGCkzsbgVq

Element: property
Metamodel: mm
xmi:id is wfP6O_iGCkzsbgVr

Element: property
Metamodel: mm
xmi:id is CrZGO_iGCkzsbhYY
```

The trace shows the applied rules (i.e. *foreign1_rl2mm*, *foreign2_er2rl* and *qualifier_mm2er*) together with the XMI ids of source elements.

In the case of the tracer, the PTL interpreter is also modified. The rules are applied backward from the target model element and each applied rule and source model element is printed.

6 Related Work

Logic programming based languages have already been explored in the context of model engineering in some works.

A first approach is [GLR+02], which describes the attempts to adopt several technologies for model transformation including logic programming. Particulary, they focused on *Mercury* and *F-Logic* logic languages. The approach [BV09] has introduced *inductive logic programming* in model transformation. The motivation of the work is that designers need to understand how to map source models to target models. With this aim, they are able to derive transformation rules from an initial and critical set of elements of the source and target models. The rules are generated in a (semi-) automatic way.

The *Tefkat* language [LR07, LS06] is a declarative language whose syntax resembles a logic language with some differences (for instance, it incorporates a *forall* construct for traversing models). In this framework, [Hea07] proposes metamodel transformations in which evolutionary aspects are formalised using the Tefkat language.

VIATRA2 [BV07] is a well-known language which has some features which make the proposal close to our approach. Although the specification of model transformation is supported by graph transformations, Prolog is adopted as transformation engine. Thus XMI models and rules are translated into a Prolog graph notation.

Prolog has been also used in the *Model Manipulation Tool (MoMaT)* [Sto07] for representing and verifying models. In [GW12], they present a declarative approach for modeling requirements (designs and patterns) which are encoded as Prolog predicates. A search routine based on Prolog returns program fragments of the model implementation. Traceability and code generation are based on logic programming. They use *JTransformer*, which is a logic-based query and transformation engine for Java code, based on the Eclipse IDE. Logic programming based model querying is studied in [DH10], in which a logic-based facts represent meta-models. In [Sch09] they study a transformation mechanism for the EMF Ecore platform using Prolog as rule-based mechanism. Prolog terms are used and predicates are used for deconstructing and reconstructing a term of a model.

In [CFLL08] they authors have compared OCL and Prolog for querying UML models. They have found that Prolog is faster when execution time of queries is linear. *Abductive logic programming* is used in [HLR09] for reversible model transformations, in which changes of the source model are computed from a given change of the target model. In [KNL11] they propose consistency checking of class and sequence diagrams based on Prolog. Consistency checking rules as well as UML models are represented in Prolog, and Prolog reasoning engine is used to automatically find inconsistencies.

On the other hand, ATL debugging has been explored in [Ecl11], that asserts debugging instructions in ATL transformations. Each time an attribute is assigned to a value, a log file is updated with the rule name, the attribute and the value. In [Jou05], they have studied traceability, that is, links between source and target models, which are instances of a traceability meta-model.

Most of the quoted works make use of Prolog for representing meta-models and model elements. The representation varies from one to another, but in essence Prolog facts are used for representing model instances while rules are used for representing transformations and constraints on models. In our case Prolog facts are also used for representing model instances however they are not directly handled by the programmer given that they are automatically generated from XMI files. Prolog rules are used as helpers in PTL but the main component of PTL are mapping rules which are rules inspired in ATL. Mapping rules are automatically translated into Prolog rules by the compiler.

In our case, debugging is intended to provide a log of the rules that cannot be applied in a certain transformation. The debugging process shows also a log with the name of the rules that cannot be applied together with the Boolean conditions and object creations that fail. The case of tracing enables to navigate from a certain target model element to the source model, that is, applied rules and source elements that contribute to the target element. In our experience using the debugging and tracing tools they are very useful to correct usual programming errors in our

language.

7 Conclusions and Future Work

In this paper we have presented a model transformation language based on logic programming. We have also described the implementation the language which consists in the encoding of mapping rules by Prolog rules. Furthermore, we have shown how to debug and trace the execution of rules. As future work, we could also extend debugging and tracing capabilities. For instance, we have considered tracing from the target model to the source model, however, tracing from source model to target model would also be possible and interesting. Finally, we would like to improve the performance of our system. The execution times we have obtained for small examples are satisfactory. However, we would like to optimize Prolog code and Prolog representation, for instance, storing Prolog facts in secondary memory for large models.

Acknowledgements: This work has been supported by the Spanish Ministry MICINN and Ingenieros Alborada IDI under grant TRA2009-0309. This work has been also supported by the EU (FEDER) and the Spanish Ministry MICINN under grants TIN2010-15588, TIN2008-06622-C03-03, and the JUNTA ANDALUCIA (proyecto de excelencia) ref. TIC-6114.

Bibliography

[BV07] A. Balogh, D. Varró. The Model Transformation Language of the VIATRA2 Framework. *Science of Programming* 68(3):187–207, October 2007.

[BV09] Z. Balogh, D. Varró. Model Transformation by Example Using Inductive Logic Programming. *Software and Systems Modeling* 8(3):347–364, 2009.

[CFLL08] J. Chimia-Opoka, M. Felderer, C. Lenz, C. Lange. Querying UML models using OCL and Prolog: A performance study. In *Software Testing Verification and Validation Workshop, 2008. ICSTW'08. IEEE International Conference on*. Pp. 81–88. 2008.

[CH06] K. Czarnecki, S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal* 45(3):621–645, 2006.

[DH10] P. Dohrmann, S. Herold. Designing and Applying a Framework for Logic-Based Model Querying. In *Software Engineering and Advanced Applications (SEAA), 2010 36th EUROMICRO Conference on*. Pp. 164–171. 2010.

[Ecl11] Eclipse. ATL to BindingDebugger. Technical report, http://www.eclipse.org/m2m/atl/atlTransformations/#ATL2BindingDebugger, 2011.

[GLR+02] A. Gerber, M. Lawley, K. Raymond, J. Steel, A. Wood. Transformation: The Missing Link of MDA. In *Procs of ICGT'02*. Pp. 90–105. LNCS 2505, Springer, London, UK, 2002.

[GW12] M. Goldberg, G. Wiener. A Declarative Approach for Software Modeling. In *Practical Aspects of Declarative Languages: 14th International Symposium, PADL 2011, Philadelphia, Pa, January 23-24, 2012. Proceedings*. Volume 7149, pp. 18–32. 2012.

[Hea07] D. I. Hearnden. *Deltaware: Incremental Change Propagation for Automating Software Evolution in Model-Driven Architecture*. PhD thesis, Centre or Institute School of Information Tech & Elec Engineering, Univ. of Queensland, 2007.

[HLR09] T. Hettel, M. Lawley, K. Raymond. Towards model round-trip engineering: an abductive approach. *Theory and Practice of Model Transformations*, pp. 100–115, 2009.

[JABK08] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev. ATL: A model transformation tool. *Science of Computer Programming* 72(1-2):31–39, 2008.

[Jou05] F. Jouault. Loosely coupled traceability for ATL. In *Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability, Nuremberg, Germany*. Volume 91, pp. 29–37. 2005.

[KNL11] Z. Khai, A. Nadeem, G. Lee. A Prolog Based Approach to Consistency Checking of UML Class and Sequence Diagrams. *Software Engineering, Business Continuity, and Education*, pp. 85–96, 2011.

[LR07] M. Lawley, K. Raymond. Implementing a practical declarative logic-based model transformation engine. In *SAC'07: Proceedings of the 2007 ACM Symposium on Applied Computing*. Pp. 971–977. ACM, New York, NY, USA, 2007. doi:http://doi.acm.org/10.1145/1244002.1244216

[LS06] M. Lawley, J. Steel. Practical Declarative Model Transformation with Tefkat. In *MoDELS Satellite Events*. Pp. 139–150. LNCS 3844, Springer, 2006.

[OMG03] OMG. MDA Spec. Technical report, http://www.omg.org/mda/specs.htm, 2003.

[Sch09] B. Schätz. Formalization and rule-based transformation of EMF Ecore-based models. *Software Language Engineering*, pp. 227–244, 2009.

[Sto07] H. Storrle. A Prolog-based Approach to Representing and Querying UML Models. In *Intl. Ws. Visual Languages and Logic (VLL'07)*. Volume 274, pp. 71–84. 2007.

[Tra05] L. Tratt. Model transformations and tool integration. *Software and System Modeling* 4(2):112–122, 2005.

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): Actas de las "XII Jornadas sobre Programación y Lenguajes (PROLE'2012), Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

Intérprete PsiXML para gramáticas de mini-Lenguajes XML en aplicaciones Web

Enrique Chavarriaga[1], Fernando Díez[1] y Alfonso Díez[2]

[1]Escuela Politécnica Superior. Universidad Autónoma de Madrid
Avda. Tomás y Valiente 11. 28049 Madrid.
jesusenrique.chavarriaga@uam.es, fernando.diez@uam.es
[2]BET Value SLR
Fuentes 10. 28013 Madrid.
adiez@betvalue.com

Resumen: El uso de lenguajes y protocolos XML es una de las herramientas de trabajo más explotadas para la creación de aplicaciones Web. Lenguajes basados en XML como ASP.NET o Java Server Face, en combinación con lenguajes robustos de programación como C# y Java, respectivamente, se emplean con frecuencia para generar páginas dinámicas en la arquitectura servidor de un sistema. Por su parte, considerado como metalenguaje, XML permite definir gramáticas como XSLT, MathML, SVG, y/o SMIL que enriquecen el modelo de presentación de la página web. Actualmente la web 2.0 nos ofrece tecnologías, servicios y herramientas que permiten construir páginas verdaderamente funcionales, agradables y usables en la arquitectura cliente. En este trabajo combinamos la definición de una gramática de lenguaje específico XML con los beneficios de un intérprete de lenguaje de programación. Introducimos el concepto de mini-lenguaje XML como aquellas gramáticas basadas en un conjunto de *tags* asociados a un modelo de diagramas de clases, evaluables sobre un intérprete especializado XML. Este intérprete puede ser usado en diferentes facetas de una aplicación Web como componente reutilizable en el sistema. Incluiremos como caso de estudio el diseño de una página Web para el manejo de diferentes fuentes Web, en particular fuentes RSS.

Palabras Clave: Intérprete de Programación, Lenguajes XML, Gramáticas de Lenguajes XML, Fuentes RSS.

1 Introducción

En las últimas décadas el desarrollo de Internet ha constituido, de algún modo, una revolución tecnológica de facto a nivel mundial. La tecnología basada en el uso de la red como plataforma de difusión del conocimiento a todos los niveles ha posibilitado un crecimiento inconmensurable de multitud de nuevas oportunidades de negocio. En este marco dinámico de cambio global, el diseño y la creación de aplicaciones Web, basadas en la construcción y presentación de páginas Web (W3C: Web Design an Applications), supone en la actualidad un reto tecnológico. Son numerosas y muy diversas las tecnologías necesarias para la creación de valor en este mercado. Las principales tecnologías están basadas en HTML para el manejo de

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): PROLE'2012, pp. 213-227, ISBN:978-84-15487-27-2.
Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

la estructura de una aplicación y en CSS para la definición del estilo y presentación (W3C: HTML & CSS, 2010).

El uso de lenguajes y protocolos XML es una de las herramientas explotadas para la creación de aplicaciones Web. Como metalenguaje, XML permite definir diferentes gramáticas como XSLT, SVG, SMIL y/o MathML, que enriquecen el modelo de presentación de la página Web (W3C: XML Technology, 2010) (W3C: Web Design an Applications). Por ejemplo, si se requiere expresar algún tipo de creatividad artística dentro de una página web, se pueden incluir gráficos dinámicos con SVG y/o PNG, o bien audio y video con SMIL y/o notación matemática con MathML. Por su parte, el uso de lenguajes script y AJAX (Holzner, 2006), agrega dinamismo a las páginas web, así como el uso de DHTML.

La web 2.0 (Marín de la Iglesia, 2010) nos ofrece tecnologías, servicios y herramientas necesarias en el cliente para construir páginas verdaderamente funcionales, agradables y usables. Además, con la web 2.0 los usuarios gestionan la información según sus necesidades: pueden agregar, modificar y/o borrar información. Cuando se usa la Web como plataforma de desarrollo, se crea un conjunto de servicios y contenidos susceptibles de ser transformados, mezclados, y/o modificada su visualización (O'Reilly, 2005).

En este trabajo buscamos combinar la definición de una gramática de lenguaje específico XML con los beneficios de un intérprete de lenguaje de programación. En consecuencia, es crear un programa escrito con gramática XML, que represente un lenguaje de alto nivel y que asociado a un modelo de clases al lado del cliente web y apoyado en la web 2.0, den solución a un problema específico dado. Para ello es necesario crear un intérprete especializado XML que analice y evalúe dichos programas y puede ser usado en diferentes facetas de una aplicación Web como componente reutilizable en el sistema.

Incluiremos como caso de estudio el diseño de una página Web para el manejo de diferentes fuentes Web, en particular fuentes RSS. Propone un tratamiento novedoso de los lenguajes basados en XML, tratados mediante un intérprete especializado y con la ventaja de no requerir para su programación el uso de ninguna tecnología del servidor, siendo posible efectuar todo el despliegue de la aplicación en el cliente, con un mínimo coste computacional.

2 Estado del arte

Los intérpretes de programación

Un **intérprete** es una aplicación que toma un **P**/**LF** (*Programa* **P** escrito en un *Lenguaje Fuente* **LF**) y datos, los analiza y evalúa simultáneamente para obtener unos resultados, ver Figura 1. Los compiladores, a diferencia de los intérpretes, transforman el **P**/**LF** a código objeto (proceso de compilación), con él lo ejecutan, toman los datos de entrada y generan los resultados (Ruiz Catalán, 2010).

En la Figura 1 se muestra el proceso y componentes principales que intervienen en un intérprete. El *Traductor* hace un primer proceso de *scanner*, consistente en la descomposición del programa fuente en componentes léxicos (*Análisis Léxico*). Posteriormente el proceso de *parsing* agrupa estos componentes en frases gramaticales (*Análisis Sintáctico*) y comprueba la validez semántica de las sentencias aceptadas por el lenguaje (*Análisis Semántico*). El resultado del traductor es la *Representación Interna* **RI** (o código intermedio) del *Lenguaje Fuente*. Los *árboles sintácticos* son los más utilizados, pero también se pueden usar las *estructuras de pila* para una mayor eficiencia en la evaluación. En caso de que existan errores

de algún tipo, estos son proporcionados como salida. Durante el proceso se genera la *Tabla de Símbolos*, que contiene información asociada al lenguaje (Mozgovoy, 2010).

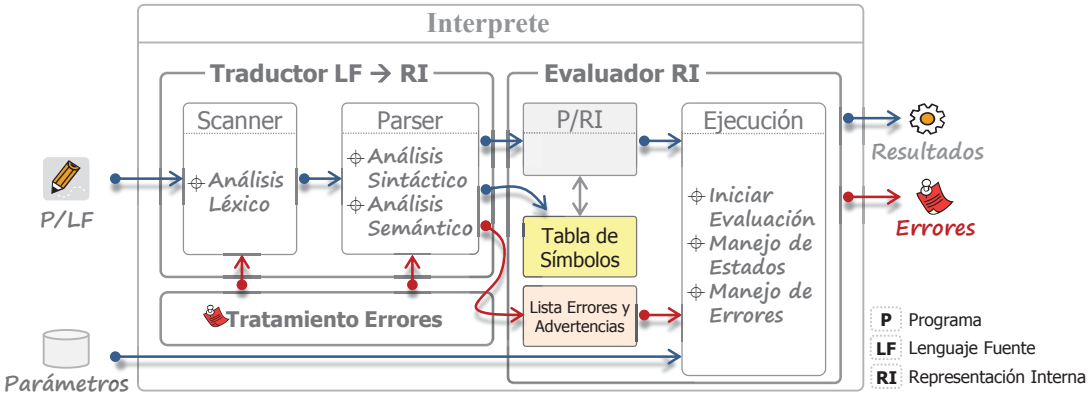


Figura 1. Intérprete de un *Programa* P escrito en un *Lenguaje Fuente* LF.

El *Evaluador* es otro de los componentes de un intérprete. Es el encargado de tomar la *Representación Interna* RI y los datos de entrada, y lleva a cabo las acciones indicadas de evaluación del programa para obtener los resultados en la salida. Durante este proceso se deben contemplar los errores de evaluación.

Los Intérpretes por lo general son sencillos de implementar aprovechando herramientas que usen Lex y Yacc (Levine, Mason, & Brown, 1992), como Flex & Bison (Levine J. , 2009) para construir el traductor. En los lenguajes interpretados, al poderse modificar y/o ampliar el lenguaje fuente a medida que se evalúa, proporcionan una mayor flexibilidad. Se incrementa su uso, si el intérprete es portable a distintas plataformas.

La Web 2.0 y los lenguajes XML

El diseño y la creación de aplicaciones Web se basan en la construcción y presentación de páginas web (W3C: Web Design an Applications) (Van Duyne, Landay, & Hong, 2006), sus principales tecnologías son HTML para el manejo de su estructura y CSS para su estilo y presentación. El uso de lenguajes script y AJAX (Holzner, 2006) (W3C: JavaScript Web APIs, 2010), agrega dinamismo a las páginas web. Lenguajes como JavaScript, basados en la especificación ECMAScript (Ecma International: ECMA-262, 2011) junto con la ayuda de la interfaz de programación de aplicaciones (API) implementada en los diferentes navegadores que proporciona el Modelo de Objetos de Documento DOM (W3C: DOM, 2004), se dota de dinamismo a las páginas, facilitando la navegación por el contenido. Así mismo estas API facilitan dotar de estructura y el estilo del documento HTML o XML, permitiendo modificar, ocultar, mover, eliminar y/o cambiar las características del mismo.

La Web 2.0 ofrece una variedad de tecnologías y servicios donde los usuarios agregan, modifican y borran información según sus necesidades (Ribes, 2007). Cuando se emplea la Web como plataforma de desarrollo, y se crea un conjunto de servicios y contenidos susceptibles a transformaciones, mezclas, cambios de visualización, y además, están diseñadas para diferentes dispositivos, se está caracterizando una aplicación Web 2.0 (O'Reilly, 2005). Algunas de las tecnologías de la Web 2.0 más notables son: Widget (Northover & Wilson, 2004), RSS, Mashup, Permalinks, REST, Ruby on Rails entre otras. Las Wikis, los Weblogs,

los Podcasts, los Videoblogs, los Servicios de RSS, son de uso más frecuentes en el mundo de Internet y forman parte de los servicios de la Web 2.0.

El uso de lenguajes y protocolos XML estándares, es otra de las herramientas de trabajo tradicionalmente más explotadas para la creación de aplicaciones Web. El lenguaje XML (W3C: XML, 2008) es un estándar para el intercambio de información estructurada y auto descriptiva entre diferentes programas, bien a nivel local o a través de las redes. Como metalenguaje, XML es un lenguaje de marcado extensible que permite definir **gramáticas de lenguajes específicos XML** (W3C: XML Technology, 2010). Entre los más utilizados están: XHTML, XSLT, SVG y/o SMIL. El lenguaje XHTML es la versión en XML de HTML, y una sus características, especialmente útil, es la posibilidad de incorporar otros lenguajes XML con su definición de espacios de nombres, para enriquecer el modelo de presentación. Por su parte, XSL, es una familia de lenguajes que permite describir cómo debe ser transformada y formateada la información XML para su presentación. El lenguaje XSLT es la que se usa para efectuar dicha transformación y requiere del lenguaje XPath para la búsqueda y obtención de información. Con la ayuda del lenguaje XLink, permite acceder a otras fuentes de información como documentos, imágenes y/o ficheros de Internet, para ampliar su contenido. Por último, XQuery es un lenguaje de consultas diseñado para buscar en colecciones de datos XML.

3 Intérprete PsiXML

El entorno de trabajo que planteamos se encuentra a caballo entre las aplicaciones web, los lenguajes Script en el lado del cliente web y la definición de gramáticas de lenguajes específicos XML. Haremos uso de las tecnologías y servicios de la web 2.0, de los lenguajes y protocolos XML, y la programación orientada a objetos para aportar soluciones a problemas de dominio específico. A este entorno de trabajo nos referiremos como *Entorno Web*.

3.1. Definición de Clases y Gramática

A continuación presentaremos la definición del conjunto de clases y la gramática asociada para un lenguaje específico XML en un *Entorno Web*.

Definición 1. Sea $\mathbb{C} = \{C_1, C_2, \dots, C_n\}$ un conjunto de clases en un *Entorno Web* que modelan la solución a un problema en un dominio específico dado. Denotaremos la instancia de una clase C_k como oC_k. En otras palabras, un objeto oC_k de la clase C_k.

En un *Entorno Web* la información se manipula a través de estructuras de datos, donde cada campo puede ser de tipo entero, carácter, fecha, objeto, arreglo, documentos DOM, funciones, elementos de página, etc. La estructura de datos la denotaremos por

$$\mathbb{D} = \{d_1 : v_1, d_2 : v_2, \dots, d_n : v_n\},$$

donde d_k es el nombre del campo y v_k es el valor del campo, para $k = 1, \dots, n$.

Definición 2. Una gramática $\mathbb{G}$ para un lenguaje específico XML se define con la tupla:

$$\mathbb{G} = \langle \mathbb{T} | t_{root} | \Delta \rangle.$$

Donde $\mathbb{T} = \{t_1, t_2, \dots, t_m\}$ es el conjunto de *tags* o etiquetas que describen el lenguaje XML, siendo t_{root} la etiqueta raíz. La estructura del lenguaje XML definida por el conjunto $\Delta =$

$\{\Delta_1, \Delta_2, \ldots, \Delta_m\}$, tal que para cada $t_k \in \mathbb{T}$, se corresponde un sólo un $\Delta_k \in \Delta$, siendo Δ_k una estructura de datos definida por:

$$\Delta_k = \{\text{CLASS: } v_C, \text{MULTIPLICITY: } v_M, \text{CHILDREN: } v_{CH}, \text{VALIDATOR: } v_V, \text{WIDGET: } v_W\}.$$

Donde los posibles valores se definen en la Tabla 1.

Tabla 1. Definición de campos de la estructura $\boldsymbol{\Delta_k \in \Delta}$ de una etiqueta $\boldsymbol{t_k \in \mathbb{T}}$ para una gramática $\mathbb{G}$.

Atributo	Descripción
CLASS	El valor v_C representa el nombre de una clase.
MULTIPLICITY	(Opcional) El valor v_M representa el número de apariciones de la etiqueta: 0..1, 1..1, 0..n, 1..n. Por defecto es 0..n.
CHILDREN	(Opcional) El valor $v_{CH} = [t_{k1}, t_{k2}, \ldots, t_{kp}]$ define la lista de etiquetas hijas, donde $t_{k1}, t_{k2}, \ldots, t_{kp} \in \mathbb{T}$. Por defecto $v_{CH} = [\,]$.
VALIDATOR	(Opcional) El valor v_V valida os atributos con el siguiente lenguaje: $\langle validator \rangle$ ::= { ["*"] $\langle attr \rangle$ ["=" $\langle default \rangle$] $\langle list \rangle$ } $\langle list \rangle$ ::= [{ $\langle deftype \rangle$ } ["|"]] $\langle deftype \rangle$ ::= { $\langle type \rangle$ "(" [$\langle param \rangle$ { ","$\langle param \rangle$ }] ")" [";"] } $\langle type \rangle$ ::= digits | integer | identifier | text | date | url | email Donde $\langle attr \rangle$ es el nombre de atributo (el atributo es obligatorio si precede "*"), $\langle type \rangle$ es el tipo de dato, $\langle param \rangle$ es el parámetro de validación para el tipo de dato y $\langle default \rangle$ el valor por defecto del atributo. A un atributo se le pueden definir varias reglas separadas por el carácter ";". El validador es ampliable según las necesidades de programa. Por ejemplo: "*_cantidad_=5:range(0, 10) | _moneda_:select(Euro, Peso) " Significa que el atributo *cantidad* es un número obligatorio entre 0 y 10, por defecto es 5, y *moneda* puede tomar los valores de Euro y Peso.
WIDGET	(Opcional) El valor de v_W es el widget asociado a la etiqueta para dar mayor funcionalidad al lenguaje.

3.2. Definición de Mini-lenguaje XML y Mini-programa XML

Para nuestro *Entorno Web* acuñaremos el termino "mini-lenguaje XML" a la definición de una gramática de lenguaje específico XML mediante un número de etiquetas o elementos, que asociados a un conjunto de clases dan solución a un problema en un dominio dado, según se formaliza en la Definición 3.

Definición 3. Un **mini-lenguaje XML** $\mathfrak{m}\mathbb{L}$, se define como la tupla:

$$\mathfrak{m}\mathbb{L} = \langle \mathbb{G} | \mathbb{C} | \Omega \rangle,$$

donde $\mathbb{G}$ es una gramática para un lenguaje específico XML, $\mathbb{C}$ un conjunto de clases y

$$\Omega = \{t_i \in \mathbb{T}, C_k \in \mathbb{C} | t_i \; se \; asocia \; a \; una \; clase \; C_k\}$$

el conjunto de asociaciones entre una etiqueta de $\mathbb{G}$ y una clase de $\mathbb{C}$. En particular denominaremos a la clase $C_{root} \in \mathbb{C}$, como *clase raíz*, a la clase asociada a la etiqueta raíz $t_{root} \in \mathbb{T}$.

Por su parte, al efecto de completar el marco de referencia de la metodología que vamos a desarrollar, planteamos la definición del concepto de "mini-programa XML".

Definición 4. Dada $\mathbb{G}$, gramática de un mini-lenguaje XML, denominaremos **mini-programa XML** a un documento XML escrito bajo la gramática $\mathbb{G}$, y lo denotaremos como $\mathfrak{m}\mathrm{P}_{xml}$. Denominaremos $\mathfrak{m}\mathbb{P} = \{\mathfrak{m}\mathrm{P}^1_{xml}, \mathfrak{m}\mathrm{P}^2_{xml}, \ldots\}$ el conjunto de todos los posibles mini-programas XML para la gramática $\mathbb{G}$.

3.3. Definición de Programa Web

Un componente software reutilizable construido a partir de un modelo de clases en un *Entorno Web*, aprovecha un tipo de estructuras $\mathbb{D}$ para recibir y manipular su información. Además, si al componente le agregamos el uso de lenguajes de dominio específico XML, para establecer su ejecución según las necesidades de resolución del problema, lo definiremos como un "Programa Web". Formalizamos ésta definición.

Definición 6. Sea $\mathbb{C}$ un conjunto de clases y $\mathbb{G}$ una gramática para un mini-lenguaje XML $\mathfrak{m}\mathbb{L}$. Se define un **Programa Web** en un *Entorno Web* denotado por P_{web}, como:

$$\mathrm{P}_{web} = \{\mathfrak{m}\mathrm{P}_{xml}, \mathbb{O}, oC_{root}, \mathbb{D}, \mathbb{W}, e\},$$

donde:

i. $\mathfrak{m}\mathrm{P}_{xml}$ es un mini-programa XML con gramática $\mathbb{G}$.
ii. El conjunto $\mathbb{O} = \{oC^i_k \in C_k | C_k \in \mathbb{C}, i \; es \; un \; entero\}$ representa la Tabla de Objetos del programa, donde oC^i_k es el objeto *i*-ésimo de la clase C_k.
iii. El objeto oC_{root} es una única instancia de la clase de raíz C_{root}.
iv. El conjunto $\mathbb{D}$ representa los parámetros de entrada al programa P_{web}.
v. El conjunto $\mathbb{W}$ representa las advertencias del programa P_{web} tanto las ocurridas en el proceso de análisis, como en el proceso de ejecución.
vi. En un programa P_{xml}, e representará su estado actual. En la Figura 2 se muestra el diagrama de estados para P_{web}, con: INIT (creado), SCANNER (lectura de $\mathfrak{m}$P), PARSE (análisis de $\mathfrak{m}$P), RUN (programa en ejecución), STOP (programa detenido) y ERROR (error en programa).

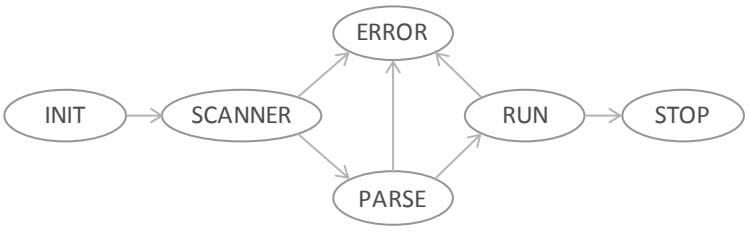


Figura 2. Diagrama de estados para un **Programa Web** en un *Entorno Web*.

3.4. Definición del Interprete PsiXML

Por el momento hemos definido los elementos necesarios para construir nuestro intérprete de mini-programas XML para un *Entorno Web*. En la Figura 3, se muestra gráficamente el Intérprete PsiXML para mini-lenguajes XML, constituido por cuatro módulos: el Traductor, el

conjunto de mini-lenguajes XML, el administrador de errores y el contexto. Basándonos en las definiciones anteriores, introducimos a continuación el concepto del Intérprete PsiXML.

Definición 7. Un **Intérprete PsiXML**, denotado por Ψ_{XML}, es el conjunto:

$$\Psi_{XML} = \{\Psi, \tau_\psi, \mathbb{K}, \mathbb{E}\}.$$

Donde:

i. $\Psi = \{\mathfrak{m}\mathbb{L}_1, \mathfrak{m}\mathbb{L}_2, \ldots, \mathfrak{m}\mathbb{L}_n\}$ es un conjunto de mini-lenguajes XML y para cada elemento $\mathfrak{m}\mathbb{L}_k = \langle \mathbb{G}_k | \mathbb{C}_k | \Omega_k \rangle$, donde $\mathbb{G}_k = \langle \mathbb{T}_k | t^k_{root} | \Delta_k \rangle$.

ii. La transformación $\langle \mathfrak{m}\mathbb{L}_k | \mathfrak{m}\mathrm{P}_{xml} | \mathbb{D} \rangle \underset{\tau_\psi}{\rightarrow} \{P_{web}, \mathbb{E}_{\mathrm{web}}\}$, donde τ_ψ toma un *mini-programa XML* $\mathfrak{m}\mathrm{P}_{xml}$ fuente escrito en $\mathfrak{m}\mathbb{L}_k$ y lo transforma en un *Programa Web* P_{web}. Donde, $\mathbb{D}$ representa el conjunto de parámetros de entrada y $\mathbb{E}_{\mathrm{web}}$ contiene el conjunto de errores, si los hay, de la transformación τ_ψ.

iii. La estructura de datos $\mathbb{K}$ se usará para el intercambio de información entre programas Web, que denominaremos **Contexto** del intérprete.

iv. $\mathbb{E}$ representa el administrador de errores del intérprete.

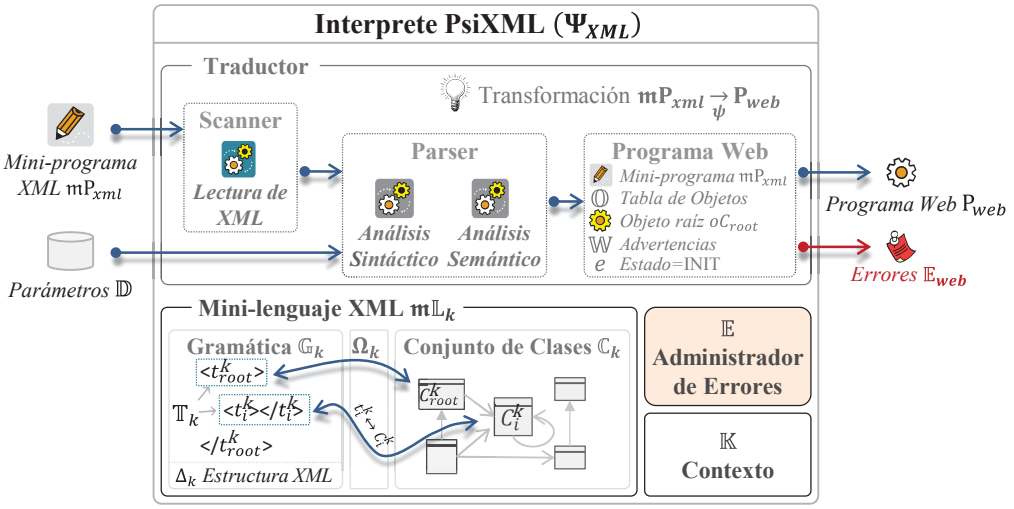


Figura 3. Intérprete PsiXML ($\boldsymbol{\Psi_{XML}}$) para mini-lenguajes XML.

En Figura 3 se observa como el **Traductor** es el encargado de realizar la transformación $\mathfrak{m}\mathrm{P}_{xml} \underset{\psi}{\rightarrow} P_{web}$ en dos pasos:

1. El **Scanner** realiza la tarea de dar lectura al mini-programa XML $\mathfrak{m}\mathrm{P}_{xml}$. Se puede hacer mediante un llamado AJAX al servidor o mediante un flujo de caracteres. En ambos casos, el cliente web es el encargado de validar si es un documento XML y obtener un documento DOM (Document Object Model).
2. El **Parser,** con el documento DOM, se recorren los elementos en forma de árbol, y se valida su estructura Δ_k de la gramática $\mathbb{G}_k$. Se obtiene la etiqueta $t^k_i \in \mathbb{T}_k$ y con ella se busca en Ω_k la clase asociada $C^k_i \in \mathbb{C}_k$, se construye el objeto oC^k_i que es registrado en la Tabla de Objetos $\mathbb{O}$ (Análisis Sintáctico). Se comprueba la validez semántica, verificando si cada objeto es aceptado por la gramática $\mathbb{G}_k$ (Análisis Semántico).

El **Administrador de Errores** en el intérprete es un mecanismo de administración de los errores y las advertencias para un programa P_{web}. En el **Traductor** hay dos tipos de errores: los que terminan la evaluación (errores de estructura del programa, de validación, falta de recursos, etc.) o aquellos que pueden ser devueltos como advertencias (conjunto $\mathbb{w}$). Por su parte, cuando un programa P_{xml} se está ejecutando solo hay errores de evaluación del programa y, dependiendo de su gravedad, pueden terminar con la misma (conjunto $\mathbb{E}$).

3.5. El Diagrama de Clases del Intérprete Ψ_{XML}

En la Figura 4, se muestra el diagrama de clases para el **Intérprete PsiXML**. Este intérprete Ψ_{XML} se implementa mediante la clase estática **PsiXML**. En dicha clase se define, a su vez, el Contexto $\mathbb{K}$ mediante la clase estática **Context** y se administra la lista de programas Web P_{web} a evaluar en la clase estática **Programs**. Para registrar un nuevo mini-lenguaje XML, $\mathfrak{m}\mathbb{L}_{\mathrm{k}} = \langle \mathbb{G}_{\mathrm{k}} | \mathbb{C}_k | \Omega_{\mathrm{k}} \rangle$ se emplea el método *registerMiniLanguageXML*. Para realizar la transformación entre el mini-lenguaje XML y el programa web $\mathfrak{m}\mathrm{P}_{xml} \underset{\psi}{\rightarrow} P_{web}$, se emplea el método *traductor_mPxmlToPweb* de la clase PsiXML.

Los mini-lenguaje XML $\mathfrak{m}\mathbb{L}_k$ son registrados usando el patrón Factory a través de la clase estática **MiniLanguageXMLFactory** y su método asociado *register*. Por su parte, una gramática $\mathbb{G}_{\mathrm{k}} = \langle \mathbb{T}_k | t_{root}^k | \Delta_k \rangle$ se construye a partir de una instancia de la clase **DefinitionElement**, con una etiqueta $t_i^k \in \mathbb{T}_{\mathrm{k}}$ y una estructura $\Delta_i^k \in \Delta_{\mathrm{k}}$. Las clases asociadas $\mathbb{C}_k$ de $\mathfrak{m}\mathbb{L}_{\mathrm{k}}$, como requisito deben heredar de la clase abstracta **Element**. Para la clase raíz, debe heredar de la clase abstracta **ElementProgram**. La clase **Element** tiene la facultad de interpretar la estructura Δ_i^k y validar su etiqueta asociada y sus atributos (hace uso de las clases **Atributes**, **Attr** y **ValidatorFactory**).

La funcionalidad del Scanner se implementa con la clase **Scanner** y la funcionalidad de un Parser, se implementa ad-hoc para cada mini-lenguaje XML, heredando de la clase abstracta **Parse**.

Finalmente, un Programa Web P_{web} se implementa con la clase **ProgramaWeb**. El mini-programa XML $\mathfrak{m}\mathrm{P}_{xml}$ se almacena en el campo *source* de tipo **DOMDocument**, la Tabla de Objetos $\mathbb{O}$ en el campo *objectsTable*, el objeto oC_{root} se obtiene con el método *rootDocument*, el conjunto $\mathbb{D}$ de parámetros en el campo *parameters*, en el conjunto de advertencias $\mathbb{W}$ se usaran objetos de la clase **Trash** (el campo *trashCompiler* contendrá los errores de compilación y el campo *trashExecute* los errores de evaluación). Por último, el campo *state* contendrá el estado actual del P_{web}. Cuando un P_{web} es evaluado con el método *execute*, el programa cambia de estado según lo descrito en la Figura 2 usando los métodos privados: *_init*, *_scanner*, *_parse*, *_run*, *_stop* y *_error*.

4 Caso de estudio

En el mundo de las noticias y la información, la difusión a través de diversos medios de comunicación es uno de los elementos claves para llegar a las personas. En el último siglo, los medios impresos, la radio y la televisión, han sido por excelencia la forma tradicional de estar al día de los acontecimientos. Con la llegada de Internet y los medios de comunicación social a través de ella, se ha revolucionado la forma de obtener y visualizar las noticias y la información. A través de la redifusión de contenidos web (de páginas web o blogs,

denominadas **fuentes Web**), los suscriptores reciben de manera actualizada la información. Con el uso de herramientas adecuadas (agregadores o lectores de fuentes web como Netvibes, Google Reader, Feed Reader, Bloglines, etc.) los usuarios pueden mantener la información centralizada y clasificada.

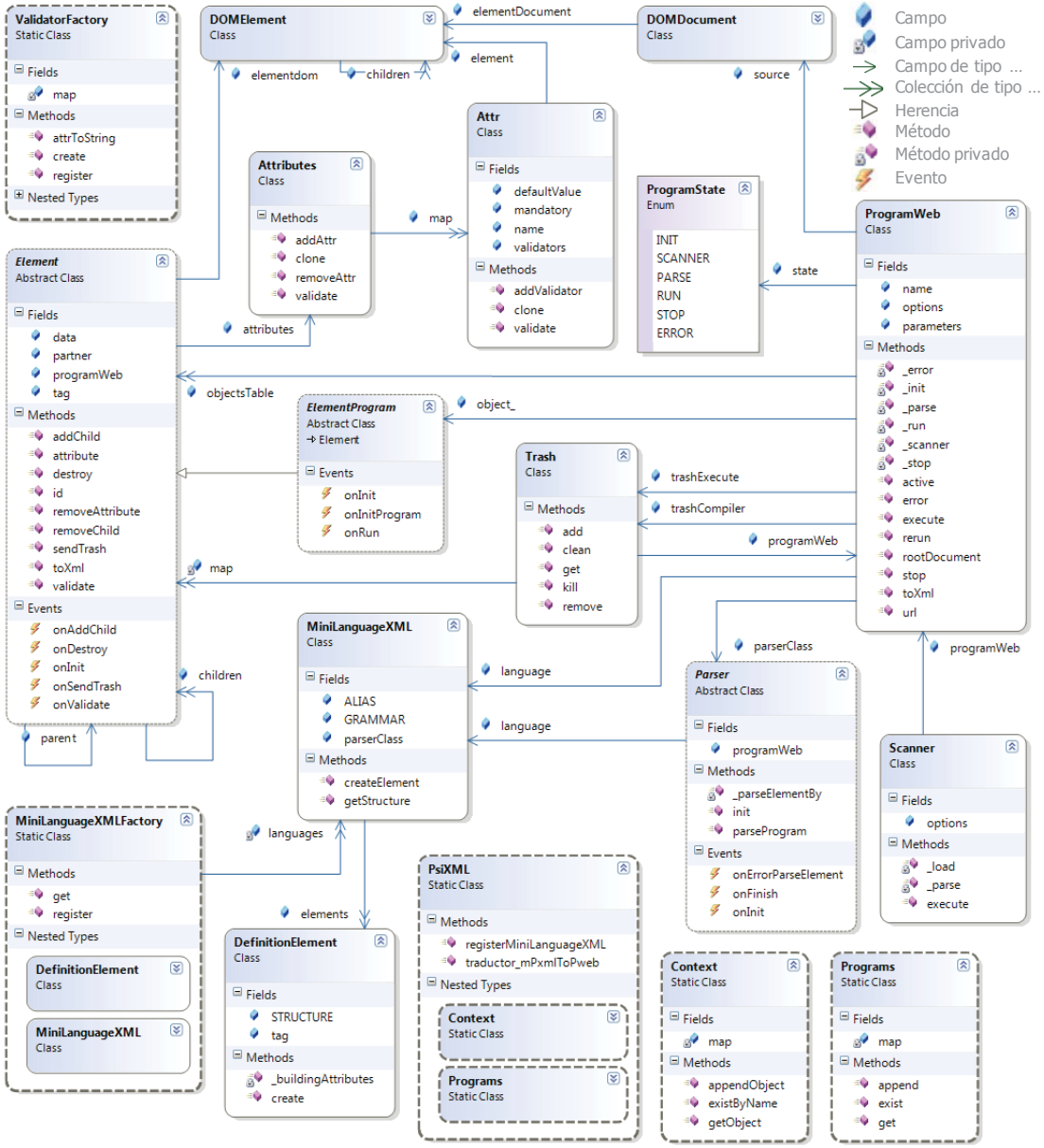


Figura 4. Diagrama de Clases para el Intérprete PsiXML (Ψ_{XML}).

Existen diferentes formatos para la gestión y visualización de las fuentes web, siendo los más utilizados RSS[1] (RSS Specifications, 2007) y Atom (IETF, 2012), ambos escritos en lenguaje

[1] Forma abreviada para una fuente web en formato RSS.

XML. El ícono se popularizó como medio de comunicación de aquellos sitios web que contienen fuentes RSS, de tal forma que las personas puedan suscribirse a los temas de su interés. El objetivo de esta sección es ilustrar el uso de un **Intérprete PsiXML** para la construcción y ejecución de una aplicación Web de difusión de noticias, a la que denominaremos **TinyViewerRSS**.

4.1. Análisis y deseño de TinyViewerRSS

En la Figura 5 se muestra el diseño de la aplicación Web **TinyViewerRSS** para el manejo de fuentes RSS. **TinyViewerRSS** gestiona fuentes RSS, permitiendo elegir, visualizar, almacenar por categorías y/o desechar noticias. Adicionalmente se muestra la gramática para el Mini-lenguaje **ViewRSS,** el cual se implementa y evalúa el Intérprete PsiXML, y que consta de una configuración (etiqueta *Configuration*), la lista de fuentes RSS (múltiples etiquetas *RSS*) y una lista de categorías (múltiples etiquetas *Category*). Cuando se lee la fuente RSS se dispone de la lista actualizada de noticias (etiqueta *News*). Las noticias, una vez etiquetadas, pueden ser visualizadas y/o a ser almacenadas en alguna de sus categorías.

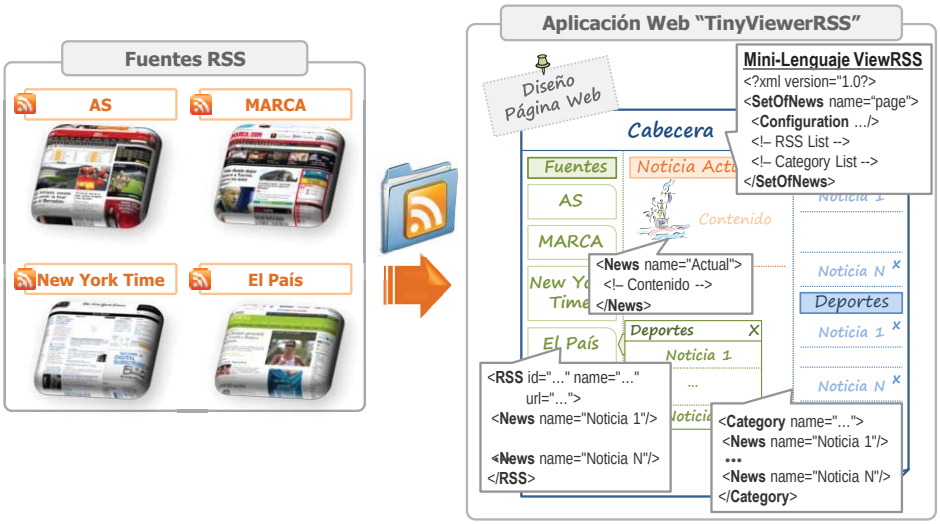


Figura 5. Diseño de una aplicación Web para la visualización de noticas a través de fuentes RSS, mediante la creación de un mini-lenguaje XML.

4.2. Mini-Lenguaje ViewRSS

La aplicación web *TinyViewerRSS* a la que acabamos de referirnos en la sección previa se construye mediante la especificación del correspondiente mini-lenguaje para la gestión y visualización de fuentes RSS, al que denominaremos **ViewRSS.** Esta sección está dedicada a la especificación de dicho mini-lenguaje XML.

Definamos la Gramática ViewRSS como:

$$\mathbb{G}_{ViewRSS} = \langle \mathbb{T}_{ViewRSS} | \text{"SetOfNews"} | \Delta_{ViewRSS} \rangle$$

Donde:

i. $\mathbb{T}_{ViewRSS} = \{\text{"SetOfNews"}, \text{"Configuration"}, \text{"RSS"}, \text{"Category"}, \text{"News"}\}$, en la Tabla 2 se muestra la descripción de uso de cada etiqueta.

ii. "$SetOfNews$" es la etiqueta raíz.

iii. $\Delta_{ViewRSS} = \{\Delta_{SetOfNews}, \Delta_{Configuration}, \Delta_{RSS}, \Delta_{Category}, \Delta_{News}\}$

Tabla 2. Conjunto de etiquetas $\mathbb{T}_{ViewRSS}$ para la gramática ViewRSS $\mathbb{G}_{ViewRSS}$.

Etiqueta	Descripción
SetOfNews	Punto de partida para la evaluación del Programa Web $P_{ViewRSS}$ (en particular para este mini-lenguaje lo denominaremos mini-programa **ViewRSS**). Contiene la configuración de su interfaz Web (etiqueta **Configuration**), la lista de fuentes RSS (múltiples etiquetas **RSS**) y la lista de categorías (múltiples etiquetas **Category**). $\Delta_{SetOfNews} = \{CLASS: \text{"SetOfNews"}, CHILDREN: [\text{"Configuration"}, \text{"RSS"}, \text{"Category"}]\}$
Configuration	Establece el título de la página (atributo *name*), el tema de la aplicación Web (atributo *theme*) y estilo de letra (atributo *font*). $\Delta_{Configuration} = \{CLASS: \text{"Configuration"}, VALIDATOR: \text{" * name: text"}]\}$
RSS	Se define mediante un identificador (atributo *id*), un nombre (atributo *name*) y la URL de la fuente RSS (atributo *url*). Contiene la lista de noticias (múltiples etiquetas **News**) de la fuente RSS una vez ha sido cargada. $\Delta_{RSS} = \{CLASS: \text{"RSS"}, VALIDATOR: \text{" * id: identifier\| * url: url"}], CHILDREN: [\text{"News"}]\}$
Category	Posee la lista de noticias almacenadas por categoría. El nombre que lo identifica está definido en el atributo *name*. $\Delta_{Category} = \{CLASS: \text{"Category"}, VALIDATOR: \text{" * name: text"}, CHILDREN: [\text{"News"}]\}$
News	Contiene la noticia obtenida de la fuente RSS y se le asigna un identificador único (atributo *id*). $\Delta_{News} = \{CLASS: \text{"News"}\}$.

Una vez definida la gramática del mini-lenguaje para el problema que deseamos tratar definimos el conjunto de clases $\mathbb{C}_{ViewRSS}$ que da solución a la gestión y visualización de fuentes RSS:

$$\mathbb{C}_{ViewRSS} = \{SetOfNews, Configuration, RSS, Category, News\}.$$

En la Figura 6 se muestra el diagrama de clases ViewRSS. En general, no es necesario tener para cada etiqueta una clase. Depende de la funcionalidad que se quiera implementar. Por ejemplo, en el caso de estudio que se presenta, se implementa el conjunto de noticias (clase **SetOfNews**) donde se añaden y eliminan tanto fuentes RSS (mediante los métodos *addRSS*, *deleteRSS,* los cuales aparecen enumerados en la descripción de la clase) como categorías (métodos *addSection*, *deleteSection*), y se modifica la configuración de la interfaz Web (método *modifyConfiguration*). Cada fuente RSS (clase **RSS**) tiene la capacidad de actualizar su contenido realizando una llamada AJAX con la URL del atributo *url* (método *refresh*), obtener una noticia (método *getNews*), y crear un menú de titulares (método *createMenu*). Por su parte, una categoría (clase **Category**) puede añadir o eliminar noticias (métodos *addNews*, *deleteNews*).

La clase **ViewNews** se implementa para visualizar las noticias elegidas de las fuentes RSS o de las diferentes categorías. Por su parte, la clase **News** define una noticia y se le asigna un identificador único (atributo *id*), en el momento de ser enviada para ser visualizada (método *sendToView*). En ese mismo instante es manipulada dinámicamente para su presentación a través de XSLT por la clase ViewNews, a través del método *view*. También puede optarse por ser enviada a una categoría (metódo *sentToCategory*).

Por su parte, el mini-lenguaje ViewRSS, además de la gramática y el conjunto de clases asociados que acabamos de describir, lleva asociado un conjunto de asociaciones denotado $\Omega_{ViewRSS}$, y definido como sigue:

$$\Omega_{ViewRSS} = \begin{Bmatrix} \text{"SetOfNews"} \rightarrow SetOfNews, \text{"Configuration"} \rightarrow Configuration, \text{"RSS"} \rightarrow RSS, \\ \text{"Category"} \rightarrow Category, \text{"News"} \rightarrow News \end{Bmatrix}$$

En este conjunto la clase SetOfNews va a constituir, por definición, la clase raíz. Una vez definidos los elementos necesarios y aplicando la Definición 3, se especifica el **Mini-lenguaje ViewRSS** denotado $\mathfrak{m}\mathbb{L}_{ViewRSS}$ mediante la tupla:

$$\mathfrak{m}\mathbb{L}_{ViewRSS} = \langle \mathbb{G}_{ViewRSS} | \mathbb{C}_{ViewRSS} | \Omega_{ViewRSS} \rangle.$$

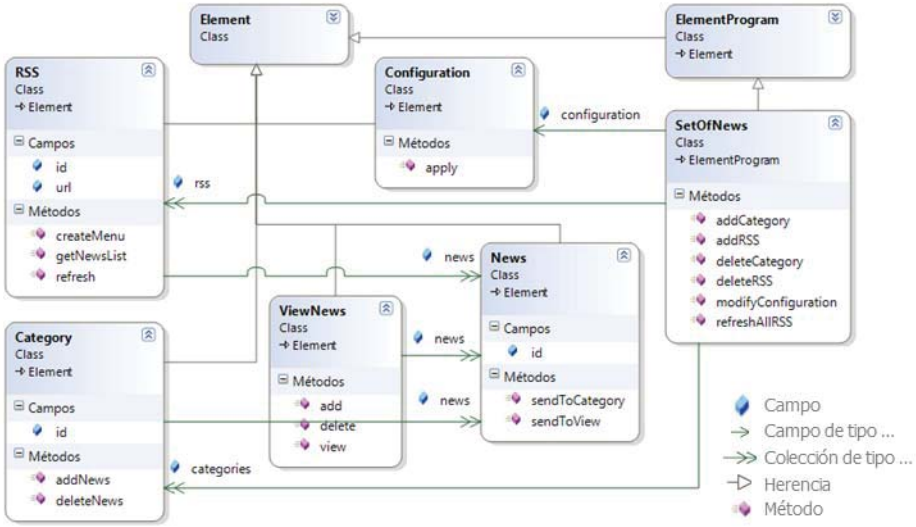


Figura 6. Diagrama de clases para el mini-lenguaje **ViewRSS**.

4.3. Ejemplo de mini-programa ViewRSS

Vamos a ilustrar la funcionalidad del mini-lenguaje $\mathfrak{m}\mathbb{L}_{ViewRSS}$ junto con el mini-programa asociado ViewRSS para resolver el problema de creación de un mini-programa donde se definan las fuentes RSS de los periódicos: *AS, MARCA, The New York Time Sport* y *El País Deportes* (ver Figura 5), con la condición particular de que no tener registradas noticias al inicio, en las categorías de Fútbol y Tenis. En la Tabla 3, se muestra el mini-programa necesario para la definición del agregador de noticias. Una vez definido el mini-lenguaje $\mathfrak{m}\mathbb{L}_{ViewRSS}$, y con el uso del **Intérprete PsiXML** podemos poner en marcha el agregador de noticias de fuentes en formato RSS, es decir, la aplicación Web *TinyViewerRSS*. Para la creación de la aplicación Web se usó jQuery User Interface[2], biblioteca de componentes visuales bajo la filosofía de jQuery.

En la Figura 7 se muestra el resultado de la aplicación Web desarrollada. La aplicación y el directorio de implementación se encuentran accesibles en los siguientes enlaces:

- PsiXML.js[3]: Implementación del Intérprete **PsiXML**.

[2] http://www.jqueryui.com

[3] http://hilas.ii.uam.es/tinyViewerRSS/js/psi/PsiXML.js

- Directorio ViewRSS[4]: Directorio de implementación del mini-lenguaje **ViewRSS**.
- Aplicación Web ***TinyViewerRSS***[5]

Tabla 3. Mini-programa **ViewRSS** para el mini-lenguaje $\mathfrak{m}\mathbb{L}_{ViewRSS}$.

Fichero: Mini-programa **ViewRSS**

```
<?xml version="1.0" encoding="utf-8" ?>
<SetOfNews>
  <Configuration name="Deportes PsiRSS" theme="default" font="arial,12px"/>
  <RSS id="as" name="AS" url="www.as.com/rss.html"/>
  <RSS id="marca" name="MARCA" url="estaticos.marca.com/rss/portada.xml"/>
  <RSS id="nytimes" name="The New York Times"
       url=" feeds1.nytimes.com/nyt/rss/Sports?format=xml"/>
  <RSS id="elpais" name="El País" url="ep00.epimg.net/rss/deportes/portada.xml"/>
  <Section id="futbol" name="Fultbol"></Section>
  <Section id="tenis" name="Tenis"></Section>
</SetOfNews>
```

Finalmente cabe hacer notar que el objetivo final del caso de uso presentado no es el de hacer una aplicación Web completamente depurada, sino mostrar cómo utilizar el concepto de mini-lenguajes XML y el uso del Intérprete PsiXML para la evaluación de mini-programas XML.

Figura 7. Aplicación Web **TinyViewerRSS**.

5 Conclusiones

En el artículo se ha presentado la combinación de la definición de una gramática de lenguaje específico XML con un intérprete especializado en ejecución de mini-programas XML, que hemos denominado **Intérprete PsiXML**. Por sí solo no tiene ningún tipo de funcionalidad, pero con la definición de un mini-lenguaje XML y sus clases asociadas forman un componente reutilizable y potente.

[4] http://hilas.ii.uam.es/tinyViewerRSS/js/viewrss/

[5] http://hilas.ii.uam.es/tinyViewerRSS/

Para validar el Intérprete PsiXML se ha diseñado una página Web para el manejo de fuentes RSS, implementando el diagrama de clases del mini-lenguaje ViewRSS. La funcionalidad de la aplicación creada se implementó ad-hoc y se obtuvo un ejemplo práctico de su uso.

El empleo de esta metodología *minimalista* desde el punto de vista de la tecnología necesaria para crear este tipo de aplicaciones web, donde no es necesario programar en lenguajes de servidor, ni mantener información en bases de datos relacionales, facilita el prototipado y desarrollo rápido de aplicaciones Web. Si a esta facilidad añadimos el uso de servicios de alojamiento de archivos multiplataforma en la nube (como iCloud, Dropbox, Cloud Power, etc.) para hospedar los mini-programas XML y Datos XML, la potencialidad de la metodología propuesta, basada en el uso del Intérperte PsiXML se incrementa notablemente. Así, haciendo un uso selectivo de las tecnologías y servicios de la Web 2.0, disponemos de las herramientas necesarias para construir aplicaciones Web altamente funcionales, robustas y usables.

En conclusión, aún es necesario desarrollar mejoras para el intérprete PsiXML mediante otros mini-lenguajes XML, así como validar su funcionamiento, agregar un contexto para la comunicación entre mini-programas XML, y darle mayor funcionalidad. En cualquier caso, el objetivo final que se persigue es el de permitir la comunicación entre dos mini-programas XML sin renunciar a la simplicidad del lenguaje.

Conscientemente nos separamos de los lenguajes de programación tradicionales como Java, C#, PHP, etc. para la creación de páginas dinámicas en el servidor, buscando nuevas perspectivas para el diseño e implementación de aplicaciones Web, basadas en intérpretes con lenguajes de dominio específico XML.

6 Trabajos futuros

Dada la sencillez del Intérprete PsiXML pensamos en plantear un lenguaje visual de dominio específico para facilitar la construcción de los mini-lenguajes XML. A partir de XML obtenemos un esqueleto de las clases asociadas del mini-lenguaje XML. Para este se puede hacer uso de las transformaciones XSLT, creando un lenguaje de especificación para el intérprete.

Otra propuesta de ampliación y mejora que planteamos es la de poder validar múltiples mini-lenguajes XML, es decir, crear dos o más mini-lenguajes XML y verificar que no produzcan algún tipo de conflicto en su ejecución. De antemano se puede pensar que la evaluación de cada mini-programa XML crea su propio programa Web pero eso depende del diseño, la implementación y la funcionalidad para cada mini-lenguaje XML.

7 Trabajos citados

Ait-Kaci, H. (1999). *Warren's Abstract Machine: A Tutorial Reconstruction* (Reprinted ed.). British Columbia, Canada: The MIT Press.

Bishop, C. (2006). *Pattern Recognition and Machine Learning.* Springer.

Ecma International: ECMA-262. (Junio de 2011). *ECMAScript Language Specification.* Recuperado el 2012, de http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf

Holzner, S. (2006). *Ajax For Dummies®.* John Wiley & Sons.

IETF. (2012). *IETF: Internet Engineering Task Force. Atom Publishing Format and Protocol*. Recuperado el Febrero de 2012, de http://datatracker.ietf.org/wg/atompub/charter/

Kolweyh, M., & Lechner, U. (2006). Towards P2P Information Systems. En *Lecture Notes in Computer Science: Innovative Internet Community Systems* (págs. 79-90). Berlin / Heidelberg: Springer.

Levine, J. (2009). *flex & bison: Text Processing Tools.* O'Reilly Media.

Levine, J. R., Mason, T., & Brown, D. (1992). *LEX & YACC.* United States of America: O'Reilly.

Marín de la Iglesia, J. (2010). *Web 2.0: Una descripción muy sencilla de los cambios que estamos viviendo* (1º ed.). La Coruña, España: Netbiblo S.L.

Mozgovoy, M. (2010). *Algorithms, languages, automata, and compilers a practical approach.* Sudbury: Jones and Bartlett Publishers.

Northover, S., & Wilson, M. (2004). *SWT: The Standard Widget Toolkit* (Vol. 1). Addison Wesley Professional.

O'Reilly, T. (9 de Septiembre de 2005). *What Is Web 2.0: Design Patterns and Business Models for the Next Generation of Software*. (O'REILLY) Recuperado el 5 de Noviembre de 2008, de http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html

Ribes, X. (Octubre-Diciembre de 2007). *La Web 2.0. El valor de los metadatos y de la inteligencia colectiva.* Recuperado el 21 de Noviembre de 2008, de http://www.campusred.net/TELOS/articuloperspectiva.asp?idarticulo=2&rev=73

RSS Specifications. (2007). *RSS: Really Simple Syndication*. Recuperado el Febrero de 2012, de http://www.rss-specifications.com/rss-specifications.htm

Ruiz Catalán, J. (2010). *Compiladores teoría e implementación.* San Fernando de Henares: RC Libros.

Van Duyne, D. K., Landay, J., & Hong, J. I. (2006). Permalinks. En *The Design of Sites: Patterns for Creating Winning Web Sites* (Second Edition ed.). Prentice Hall.

W3C: DOM. (7 de Marzo de 2004). *Document Object Model (DOM) Level 3 Core Specification.* Obtenido de http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/

W3C: JavaScript Web APIs. (2010). *DOM (Document Object Model), XBL (XML Binding Language), WAI-ARIA (Acessible Rich Internet Applications) and Mobile Web Application.* Obtenido de http://www.w3.org/standards/webdesign/script

W3C: Web Design an Applications. (s.f.). *HTML & CSS, Scripting and Ajax, Graphics, Audio and Video, Accessibility, Internationalization, Mobile Web, Privacy and Math on the Web*. Recuperado el Febrero de 2012, de http://www.w3.org/standards/webdesign/

W3C: XML. (26 de Noviembre de 2008). *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. Recuperado el 2012, de http://www.w3.org/TR/xml/

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): Actas de las "XII Jornadas sobre Programación y Lenguajes (PROLE'2012), Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

Sesión 7

Lógica Temporal/Model checking

Chair: *Dra. Alicia Villanueva*

Sesion 7: Lógica Temporal/Model checking
Chair: Dra. Alicia Villanueva

Jose Gaintzarain, Montserrat Hermo, Paqui Lucio, Marisa Navarro and Fernando Orejas. *Invariant-Free Clausal Temporal Resolution*.

Damián Adalid, Alberto Salmerón, María del Mar Gallardo and Pedro Merino. *Testing Temporal Logic on Infinite Java Traces*.

Laura Panizo and María del Mar Gallardo. *Analyzing Hybrid Systems with JPF*.

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): Actas de las "XII Jornadas sobre Programación y Lenguajes (PROLE'2012), Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

Invariant-Free Clausal Temporal Resolution

J. Gaintzarain[1], M. Hermo[2], P. Lucio[2], M. Navarro[2] and F. Orejas[3]

[1] The University of the Basque Country, 48012-Bilbao, Spain.
[2] The University of the Basque Country, 20080-San Sebastián, Spain.
[3] Technical University of Catalonia, 08034-Barcelona, Spain.

Abstract: We provide an extended abstract of the paper with the same title and authors that is going to appear in Journal of Automated Reasoning (Online from December 2th, 2011).

Keywords: Temporal Logic, Resolution, Invariant-free, Clausal Normal Form.

Temporal logic plays a significant role in computer science, since it is an ideal tool for specifying object behaviour, cooperative protocols, reactive systems, digital circuits, concurrent programs and, in general, for reasoning about dynamic systems whose states change over time. Propositional Linear-time Temporal Logic (PLTL) is one of the most widely used temporal logics. This logic has, as the intended model for time, the standard model of natural numbers. Different contributions in the literature on temporal logic show its usefulness in computer science and other related areas. For a recent and extensive monograph on PLTL techniques and tools, we refer to [6], where the reader can find sample applications along with references to specific work that uses this temporal formalism to represent dynamic entities in a wide variety of fields. The minimal language for PLTL adds to classical propositional connectives two basic temporal connectives $\circ$ ("next") and $\mathscr{U}$ ("until") such that $\circ p$ is interpreted as "the next state makes p true" and $p \mathscr{U} q$ is interpreted as "p is true from now until q eventually becomes true". Many other useful temporal connectives can be defined as derived connectives, e.g. $\diamond$ ("eventually"), $\Box$ ("always") and $\mathscr{R}$ ("release").

Automated reasoning for temporal logic is a quite recent trend. In temporal logics, as well as in the more general framework of modal logic, different proof methods are starting to be designed, implemented, compared, and improved. Automated reasoning for PLTL, and related logics, is mainly based on tableaux and resolution. In this paper, we deal with clausal resolution for PLTL. The earliest temporal resolution method [1] uses a non-clausal approach, hence a large number of rules are required for handling general formulas instead of clauses. There is also early work (e.g. [2, 4]) related to clausal resolution for (less expressive) sublogics of PLTL. The language in [2] includes no eventualities, whereas in [4] the authors consider the strictly less expressive sublanguage of PLTL defined by using only $\circ$ and $\diamond$ as temporal connectives. The early clausal method presented in [8] considers full PLTL and uses a clausal form similar to ours, but completeness is only achieved in absence of eventualities (i.e. formulas of the form $\diamond \varphi$ or $\varphi \mathscr{U} \psi$). More recently, a fruitful trend of clausal temporal resolution methods, starting with the seminal paper of M. Fisher [5], achieves completeness for full PLTL by means of a specialized *temporal resolution* rule that needs to generate an invariant formula from a set of clauses that behaves as a loop. The methods and techniques developed in such an approach have been successfully adapted to Computation Tree Logic (CTL) (see [3]), but invariant handling seems to be a handicap for further extension to more general branching temporal logics such

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): PROLE'2012, pp. 231-232, ISBN:978-84-15487-27-2.
Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

as Full Computation Tree Logic (CTL*). In this paper, we introduce a new clausal resolution method that is sound and complete for full PLTL. Our method is based on the dual methods of tableaux and sequents for PLTL presented in [7]. On this basis we are able to perform clausal resolution in the presence of eventualities avoiding the requirement of invariant generation. We define a notion of *clausal normal form* and prove that every PLTL-formula can be translated into an equisatisfiable set of clauses. Our resolution mechanism explicitly simulates the transition from one world to the next one. Inside each world, we apply two kinds of rules: (1) the resolution and subsumption rules and (2) the fixpoint rules that split a clause with an eventuality atom into a finite number of new clauses. We prove that the method is sound and complete. In fact, it finishes for any set of clauses deciding its (un)satisfiability, hence it gives rise to a new decision procedure for PLTL. We also compare our approach with the methods in [4, 1, 8, 5].

Acknowledgements: This work has been partially supported by the Spanish Project TIN2007-66523 and the Basque Project LoRea GIU07/35.

Bibliography

[1] Martín Abadi and Zohar Manna. Nonclausal temporal deduction. In Rohit Parikh, editor, *Logic of Programs*, volume 193 of *Lect. Notes in Computer Sci.*, pages 1–15. Springer, 1985.

[2] Marianne Baudinet. Temporal logic programming is complete and expressive. In *Proceedings, Sixteenth Annual ACM Symposium on Principles of Programming Languages (POPL), Austin, Texas*, pages 267–280, 1989.

[3] Alexander Bolotov and Michael Fisher. A clausal resolution method for CTL branching-time temporal logic. *Journal of Experimental & Theo. Artif. Intell.*, 11(1):77–93, 1999.

[4] Ana R. Cavalli and Luis Fariñas del Cerro. A decision method for linear temporal logic. In Robert E. Shostak, editor, *7th International Conference on Automated Deduction, Napa, California, USA, May 14-16, 1984, Proceedings*, volume 170 of *Lecture Notes in Computer Science*, pages 113–127. Springer, 1984.

[5] M. Fisher. A resolution method for temporal logic. In John Mylopoulos and Raymond Reiter, editors, *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI), Sydney, Australia.*, pages 99–104, 1991.

[6] Michael Fisher. *An Introduction to Practical Formal Methods Using Temporal Logic*. John Wiley & Sons, Ltd, June, 2011.

[7] Jose Gaintzarain, Montserrat Hermo, Paqui Lucio, Marisa Navarro, and Fernando Orejas. Dual systems of tableaux and sequents for PLTL. *Journal of Logic and Algebraic Programming*, 78(8):701–722, 2009.

[8] G. Venkatesh. A decision method for temporal logic based on resolution. In S. N. Maheshwari, editor, *Foundations of Software Technology and Theoretical Computer Science, Fifth Conference, New Delhi, India, December 16-18, 1985, Proceedings*, volume 206 of *Lecture Notes in Computer Science*, pages 272–289. Springer, 1985.

Testing Temporal Logic on Infinite Java Traces

Damián Adalid, Alberto Salmerón, María del Mar Gallardo and Pedro Merino[1]*

[1](damian,salmeron,gallardo,pedro)@lcc.uma.es
Dpto. de Lenguajes y Ciencias de la Computación
University of Málaga
Spain

Abstract: This paper summarizes an approach for testing reactive and concurrent Java programs which combines model checking and runtime monitoring. We use a model checker for two purposes. On the one hand, it analyzes multiple program executions by generating test input parameters. On the other hand, it checks each program execution against a linear temporal logic (LTL) property. The paper presents two methods to abstract the Java states that allow efficient testing of LTL. One of this methods supports the detection of cycles to test LTL on potentially infinite Java execution traces. Runtime monitoring is used to generate the Java execution traces to be considered as input of the model checker. Our current implementation in the tool TJT uses Spin as the model checker and the Java Debug Interface (JDI) for runtime monitoring. TJT is presented as a plug-in for Eclipse and it has been successfully applied to complex public Java programs.

Keywords: testing, model checking, runtime monitoring

1 Introduction

This paper is a summary of [ASGM12] where we present a new method to convert a Java execution trace into a sequence of states that can be analyzed by the model checker Spin [Hol03]. As far as Spin implements the analysis of LTL formulas by translation to Büchi automata, we can check the formulas on Java programs with potential cycles. The Spin stuttering mechanism to deal with finite execution traces allows us to analyze any kind of program without redefining the semantics of LTL.

Our conversion of Java traces into Spin oriented traces is based on two efficient abstraction methods of the full state of the program. The so-called *counter projection* abstracts the Java state by preserving the variables in the LTL formula and adding a counter to make each state unique. As long as we do not keep all the information, the counter projection is very efficient at the price of being useful only for a limited subset of LTL. The *hash projection* abstracts each Java state with the variables in the formula plus a hash of the whole state. The way of constructing the hash makes negligible the probability of conflict for two different states, so we can trust in the Spin algorithm to check LTL based on cycle detection.

* Work partially supported by projects P07-TIC-03131 and WiTLE2.

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): PROLE'2012, pp. 233-234, ISBN:978-84-15487-27-2.
Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

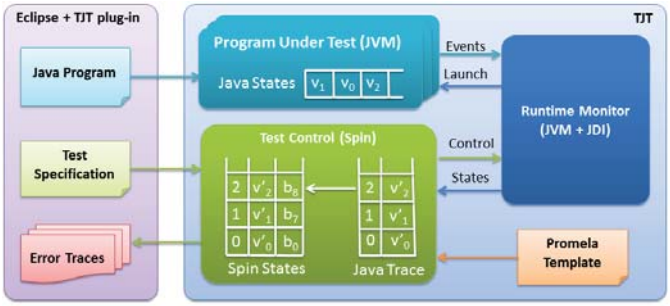


Figure 1: TJT architecture and workflow.

2 Architecture and features

Figure 1 shows an overview of this architecture and the workflow of the tool. The programmer must supply two inputs in this workflow: the Java program being tested and a test specification. The latter provides the relevant information for carrying out the tests, including the LTL property that will be checked against execution traces of the program.

TJT is divided in two modules, plus an Eclipse plug-in[1]. The test control module, implemented using the model checker Spin, prepares the test and generates a series of test inputs for the Java program as instructed by the test specification. A series of Java Virtual Machines are then launched to test the execution of the program under each generated test input. The executions are controlled by the monitoring module, which detects and reports the events that are relevant to check the LTL formula. Spin processes the information reported by the monitoring module for each execution of the program, and checks if the LTL property stated in the test specification is violated. When a trace does not satisfy a formula, it is stored to allow the user to inspect it.

Regarding related work, JPF [HP00] is a well-known model checker for Java programs. However, support for LTL formulas in JPF is still work in progress.

In conclusion, TJT is useful to test functional properties expressed as LTL on both sequential and concurrent programs with finite and infinite execution traces. The formalization of the abstraction approach guarantees the preservation of the results, and the experimental results confirm the applicability to realistic software.

Bibliography

[ASGM12] D. Adalid, A. Salmerón, M. d. M. Gallardo, P. Merino. Testing Temporal Logic on Infinite Java Traces. In *MSVVEIS 2012*. P. To appear. 2012.

[Hol03] G. J. Holzmann. *The SPIN Model Checker : Primer and Reference Manual*. Addison-Wesley Professional, September 2003.

[HP00] K. Havelund, T. Pressburger. Model Checking JAVA Programs using JAVA PathFinder. *STTT* 2(4):366–381, 2000.

[1] The plug-in and examples are available from http://www.lcc.uma.es/~salmeron/tjt/

Analyzing Hybrid Systems with JPF

Laura Panizo[1], **María-del-Mar Gallardo**[2]

[1] laurapanizo@lcc.uma.es
[2] gallardo@lcc.uma.es
University of Málaga, Málaga

Abstract: Hybrid systems are characterized by combining discrete and continuous behaviors. Verification of hybrid systems is, in general, a difficult task due to the potential complexity of the continuous dynamics. Currently, there are different formalisms and tools which are able to analyze specific types of hybrid systems, model checking being one of the most used approaches. In this paper, we propose an extension of the discrete model checker Java Path Finder in order to analyze hybrid systems. We apply a general methodology which has been successfully used to extend SPIN. This methodology is non-intrusive, and uses external libraries, such as Parma Polyhedra Library, to abstract the continuous behavior of the hybrid system independent to the underlying model checker.

Keywords: Hybrid systems, model checking, JPF

1 Introduction

A hybrid system is a system whose behavior is determined by a combination of different variables: a set of magnitudes, that follow some *continuous dynamics* which may instantaneously change, and a set of discrete variables, that behave as usual in discrete systems. Thus, hybrid systems present continuous and discrete behaviors which are closely related. On the one hand, magnitudes are modeled by the so-called continuous variables the values of which evolve over time following ordinary differential equations (ODE). On the other hand, each discrete state, also called location, can define a different dynamic for the continuous magnitudes. Thus, when a transition between two locations occurs, the dynamics of some continuous variables may change.

Hybrid systems have been used as a model for different applications, such as embedded automotive controllers [BBB+00] or manufacturing systems [Feh99]. Since many of these applications are considered *critical*, there is a general interest in the fields of design and analysis of hybrid systems. In the context of computer science, it is common to use hybrid automata [Hen96] to represent hybrid systems. Hybrid automata extend the notion of automata by adding continuous variables to discrete states whose values may continuously evolve following an ODE.

Since the algorithmic analysis of systems described with general ODEs is difficult, it is normal to constrain the equations that describe the continuous dynamics and to develop analytical methods for specific types of hybrid systems. In [ACH+95], the authors describe the *linear hybrid automata*, whose continuous variables evolve following constant differential equations. Linear hybrid automata include systems such as *timed automata* [AD91], the variables of which are clocks that move linearly over time, or *multirate timed systems*, in which continuous variables are clocks that evolve at different rates. *Rectangular hybrid automata* [Kop96] are another class

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): PROLE'2012, pp. 235-249, ISBN:978-84-15487-27-2.
Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

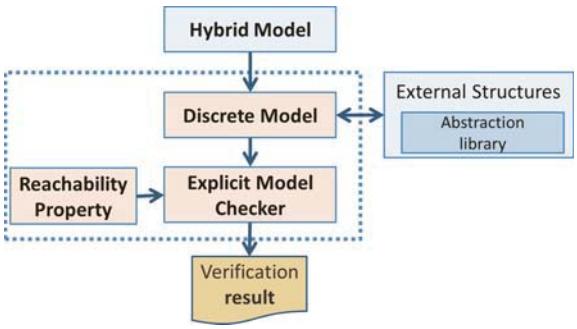


Figure 1: Architecture of the approach

of hybrid automata that define the evolution of continuous variables by $\mathbb{R}^n$-intervals. The *initialized rectangular hybrid automata* (IRHA) are a special case that forces the reset (initialization), in discrete transitions, of those continuous variables the continuous dynamics of which change.

In previous work [GP10], we proposed a methodology to extend explicit discrete model checkers to analyze IRHA in a non-intrusive way. Firstly, we chose this class of hybrid automata because their dynamics can naturally approximate more general dynamics [HHW98]. In addition, IRHA form a maximal class of hybrid automata for which model checking is decidable [HKPV98].

Figure 1 shows our methodology. Our claim is that the abstraction of the continuous behavior may be integrated in an off-the-shelf model checker in a transparent manner. To achieve this integration, we use existing libraries, such as Parma Polyhedra Library (PPL), to carry out the abstraction, and we define a set of external structures to store the necessary data, which are partially visible to the model checker. The visible part is a discrete reference to the continuous behavior which is included in the discrete section of system states handled by the model checker. Consequently, the model checker is able to analyze hybrid systems as if they were discrete. One of the advantages of this methodology is its modularity. We could replace the external library that performs the abstraction, and the main parts of the tool architecture do not have to be modified. Another advantage is that users of a given model checking tool may easily use an extension of its preferred tool able to analyze hybrid systems.

In [GP10] we reported our success in using this methodology to extend the SPIN model checker. To demonstrate the applicability of the methodology, now we use a similar approach to extend JPF. There are many differences between SPIN and JPF, that makes an adaptation of our methodology necessary. The main difference is that JPF generates a state only when there is non-deterministic behavior. In general, a change in a Java variable does not generate a JPF state. In contrast, SPIN produce states when there is any change in the variables, or the program counter. This is mainly because Java is a full programming language while PROMELA is only for modeling purposes. In addition, since Java is an object oriented language the mechanism to model an automaton also changes, from process to objects. Finally, each tool provides a different mechanism to interact with external libraries. SPIN allows the execution of native C code embedded in the model. In contrast, JPF provides an interface to execute Java bytecodes in a host Java Virtual Machine.

The paper is organized as follows. Section 2 introduces the following topics: JPF, hybrid automata and PPL. Section 3 presents the underlying formalization of our approach. Section 4

explains different aspects of the current implementation using JPF. Finally, Section 5 presents some conclusions and future work.

2 Preliminaries

2.1 Java Path Finder

Java Path Finder [jpf12] is a highly customizable execution environment for the verification of Java bytecode programs. JPF was originally designed as a model checking tool for Java but, currently, JPF provides much more functionality. The basic component of JPF is *jpf-core*, which is a special virtual machine that executes Java programs to find defects in their behavior. These defects are expressed by means of properties.

Figure 2 shows the architecture of JPF. Since JPF is a Virtual Machine running over the JavaVirtual Machine (JVM), the input of JPF is a standard Java program with some annotations and some configuration files containing the properties to be analyzed. The result of the analysis is a report that states whether the properties hold. The architecture includes two interfaces, the Model Java Interface (MJI), and the Java Native Interface (JNI). The MJI provides access to the JPF applications, while JNI gives access to the JVM host. There are three main reasons to execute bytecodes in the JVM host instead of JPF:

- To intercept native methods that JPF is forced to ignore.
- To intercept methods that would affect JPF internal class, object and thread model. MJI can also be used to extend the functionality of JPF without changing its implementation.
- To reduce state space since execution in the JVM host is non-tracked.

JPF implements a mechanism to intercept method invocations and delegates them, by means of Java reflection, to a dedicated class. This mechanism uses two types of classes: the *model class* and the *native peer class*. The model class is executed by JPF and might be completely unknown to the JVM host. This model class contains the methods that have to be intercepted. The native peer class is executed in the JVM host, and contains the implementation of the intercepted methods.

JPF implements different model checking algorithms, from the depth first search to heuristic search. All of them are based on identifying points in the program (called execution choices) where execution could proceed differently, and then systematically exploring all the possibilities. This means that theoretically JPF executes all paths through the program, not just one like a normal JVM. Typical choices are different scheduling sequences or random values, but JPF allows us to also introduce new types of choices such as user inputs or state machine events. Observe that JPF generates a new state only when an execution choice is reached, and not when a variable changes its value which is slightly different from other model checkers such as SPIN.

The jpf-core can be configured to detect different errors in the program. Some non-functional properties, such as deadlocks or unhandled exceptions, do not have to be specified. In addition, functional properties such as reachability may be detected by means of *listeners*, which are little plug-ins that monitor the execution of the program.

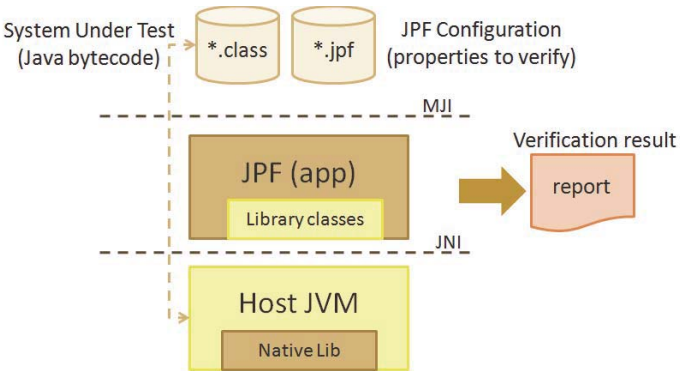


Figure 2: JPF architecture

Finally, it is worth highlighting some characteristics of JPF. First, JPF is a virtual machine running over a standard JVM, thus the execution of a Java program is slower than in the original JVM. In addition, this speed is even slower because JPF executes all possible paths of the program, in comparison with the standard JVM. Secondly, JPF cannot execute Java libraries that make use of native code, mainly because native calls, such as system calls, are difficult to revert. To solve this problem, we can use the *native peers* and *model classes* described above. The jpf-core includes implementations of some common standard Java classes, such as `java.lang.Thread`.

2.2 Hybrid Automata

Modeling a hybrid system requires using both discrete variables, which evolve following the usual discrete dynamics of computer systems, and continuous variables that change following the continuous dynamics over time described by means of ODEs. The evolution of discrete variables is assumed to be instantaneous. However, in order to describe the behavior of continuous variables it is necessary to introduce time related aspects into the model. *Hybrid automata* [Hen96] provide a formal model for hybrid systems. Hybrid automata formalize the notion of *trajectory* as a particular execution of a hybrid system in which both discrete and continuous transitions may be interleaved. In this section, we introduce this formal model [Hen96] and illustrate its behavior with an example.

Definition 1 (Hybrid automaton) A *hybrid automaton H* is a tuple $\langle Loc, T, \Sigma, X, Init, Inv, Flow, Jump \rangle$ where:

- Loc is a finite set of discrete states (locations) of the automaton.
- $T \subseteq Loc \times Loc$ is a finite set of transitions.
- Σ is the set of event names, equipped with a labeling function $lab : T \to \Sigma$.
- $X = \{x_1, \cdots, x_n\}$ is a finite set of real-valued variables. The set $\dot{X} = \{\dot{x}_1, \cdots, \dot{x}_n\}$ represents the first derivative of the elements of X during the continuous evolution in a location. In

addition, the set $X' = \{x'_1, \cdots, x'_n\}$ corresponds to updates of variables when a transition $t \in T$ takes place.

- *Init*, *Inv* and $Flow$ are functions that assign predicates to each location. Thus, $Init(loc)$ defines the possible initial values of the variables in X. $Inv(loc)$ gives the conditions on variables in X that must be true while the system stays at location loc. $Flow(loc)$ is a predicate over $X \cup \dot{X}$ that defines the evolution of the variables in loc.

- The function *Jump* assigns each transition $t \in T$ to the guards that enable the transition and to the updates of variables before transiting to the new location.

Let $[X \to \mathbb{R}]$ be the set of all maps from X to $\mathbb{R}$. If p is a predicate over X, then $[[p]]$ denotes all functions $v \in [X \to \mathbb{R}]$ that satisfy p. Similarly, we define the meaning of a predicate over X', $X \cup X'$ or $X \cup \dot{X}$. Finally, the same notion may be extended to sets of predicates such as $Init, Inv$ or $Jump$.

A hybrid state of H is a pair (loc, v), where $loc \in Loc$ is a location of the automaton, and $v \in [X \to \mathbb{R}]$ is a valuation of variables such that $v \in [[Inv(loc)]]$. A hybrid automaton behaves like a *timed transition system*. Intuitively, given $loc, loc' \in Loc$, two valuations $v, v' \in [X \to \mathbb{R}]$, and $\delta \in \mathbb{R}_{\geq 0}$, $(loc, v) \to_\delta (loc, v')$ means that the continuous variables evolve from v into v' in δ time units, and $(loc, v) \xrightarrow{\sigma}_d (loc', v')$ means that the hybrid state (loc, v) evolves instantaneously to (loc', v').

Definition 2 (Trajectories) Given a hybrid automaton $H = \langle Loc, T, \Sigma, X, Init, Inv, Flow, Jump \rangle$, transitions of H are either:

- Discrete transitions: given $t = (loc, loc') \in T$ and $lab(t) = \sigma$, $(loc, v) \xrightarrow{\sigma}_d (loc', v')$, iff $v, v' \in [X \to \mathbb{R}]$, and $(v, v') \in [[Jump(t)]]$. To simplify the notation, we drop the label σ from the discrete transitions and use simply $\to_d$.

- Continuous transitions: for each $\delta \in \mathbb{R}_{\geq 0}$, $(loc, v) \to_\delta (loc, v')$ iff there exists a differentiable function $f_v : [0, \delta] \to \mathbb{R}^n$, $\dot{f}_v : [0, \delta] \to \mathbb{R}^n$ being its first derivative, such that:

  - $f_v(0) = v, f_v(\delta) = v'$.
  - $\forall r \in [0, \delta], f_v(r) \in [[Inv(loc)]]$ and $(f_v(r), \dot{f}_v(r)) \in [[Flow(loc)]]$.

Thus, a **trajectory** is a (possible infinite) sequence of *hybrid states* such as $(loc_0, v_0) \to_{\lambda_0} \cdots (loc_j, v_j) \to_{\lambda_j} \cdots$, where $v_i \in [[Inv(loc_i)]]$, $i = 0, 1 \cdots$, $\lambda_i \in \mathbb{R} \cup \{d\}$ and $v_0 \in Init(loc_0)$.

This definition of trajectory is very intuitive, since it explicitly shows the continuous and discrete evolution of a system, leaving undetermined the time when discrete transitions take place. In addition, a state stores a unique real value for each continuous variable, which is very reasonable from a theoretical point of view. However, it is clear that the definition gives no guidance whatsoever on how to manage and analyze the non-enumerable set of possible states generated by the system.

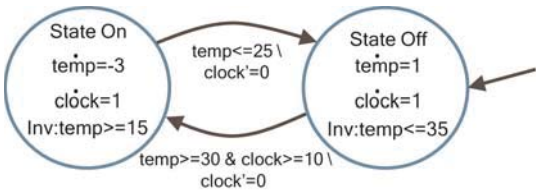


Figure 3: Hybrid automaton of fan controller

2.2.1 An Example: The Fan Controller

To illustrate the behavior of hybrid automata, we use the example of a fan controller, shown in Figure 3. The main task of this controller is to keep the temperature of the room between 15 and 35 degrees. The automaton has two locations that represent the possible modes of the fan (`On` or `Off`). There are two continuous variables: `clock`, a clock that measures the time elapsed from the last mode switching, and `temp`, which saves the room temperature. The automaton can be in location `Off` if the temperature is below 35 degrees. In this location, the first derivative of both variables is equal to one; in other words, they evolve linearly over time and at the same rate. The automaton can be in location `On` if the temperature is over 15 degrees. In this case, the first derivative of `temp` is minus three, and the derivative of `clock` is one. The transition from location `Off` to `On` is enabled when the temperature is over 30 degrees and at least 10 time units have elapsed since reaching the location `Off`. When the transition is taken, `clock` is set to zero (reset). The transition from `On` to `Off` is enabled if the temperature is below 25 degrees, and again, `clock` is reset to zero. The uncertainty of discrete transition execution generates different trajectories that mix the continuous evolution of the variables with the discrete transitions. For instance, if `temp` is initially set to 14 degrees and `clock` is set to 0, a valid trajectory of the automaton is $\langle Off,(0,14)\rangle \longrightarrow_{20} \langle Off,(20,34)\rangle \longrightarrow_{d} \langle On,(0,34)\rangle \longrightarrow_{4} \langle On,(4,22)\rangle \longrightarrow_{d} \langle Off,(0,22)\rangle \longrightarrow \cdots$ but, of course, the automaton allows many more possible trajectories.

2.3 Reachability using convex polyhedra

We use numerical abstractions to over-approximate the continuous behavior of hybrid systems. This approach has been proposed many times using different types of abstract domains such as hyperrectangles [BMP99] or convex polyhedra [CK98]. The abstraction used may restrict the equations that define the behavior of the continuous variables.

There are many open libraries that provide numerical abstractions [KMP+95, Jea02, LCW+02]. These libraries can greatly reduce the cost of developing a new tool. We use the Parma Polyhedra Library [BHZ08] to represent the continuous behavior of IRHA by means of convex polyhedra. The main features of PPL are precise arithmetic, efficiency, dynamic memory allocation, and portability (PPL is written in standard C++, with C, Java, Objective CAML and Prolog interfaces).

3 Formalization

In this section, we formalize the methodology proposed in the paper. In particular, we show how the trajectories allowed by hybrid automaton can be abstracted by discrete paths which can be analyzed later by non-hybrid off-the-shelf model checkers. Here, we formally describe this trajectory abstraction, and prove some correctness results which guarantee the soundness of our approach. To this end, we assume that the continuous and discrete transition relations, described in the trajectory definition (Definition 2), are simulated by two *abstract* transition relations. Parts of these theoretical results are well-known ([ACH+95, Ho95, HKPV98, HK99]), but we describe them here to illustrate the proposed methodology. Additionally, we prove the preservation of properties between the abstract trajectories analyzed and the concrete trajectories allowed by the initial automaton. These results provides a way to interpret the answers obtained when the abstract trajectories are analyzed by model checking.

Note that the particular abstraction of states is not relevant in the discussion that follows. However, we assume that states are abstracted using convex polyhedra according to the implementation presented in the paper. In addition, we consider that the hybrid automaton under analysis is rectangular, i.e. all automaton predicates are constraints such as $x \in (0,5]$. We start from a hybrid automaton (that could be the product of different automata [Fre05]) with n continuous variables $X = \{x_1, \cdots, x_n\}$.

Definition 3 (Abstract states) An *abstract state* is a pair $\langle loc, P \rangle$ such that $loc \in Loc$ and P is a convex polyhedron in $\mathbb{R}^n$ which satisfies the invariant $[[Inv(loc)]]$, that is, $P \subseteq [[Inv(loc)]]$. The abstract state $\langle loc, P \rangle$ is an *abstraction* of a hybrid state (loc, v) iff $v \in P$.

An *extended abstract state* is a 3-tuple $\langle loc, P, P' \rangle$, where $\langle loc, P \rangle$ is an abstract state, P' is a convex polyhedron in $\mathbb{R}^n$ and $P \subseteq P'$.

The extended abstract state $\langle loc, P, P' \rangle$ extends the abstract state $\langle loc, P \rangle$ with a third component P' that stores the polyhedron accumulated during the execution of a number of continuous transitions in an *abstract* trajectory.

The abstraction relation between hybrid and abstract states allows the simulation of the trajectories of the hybrid automaton H with sequences of abstract states. Given transition relations $\rightarrow_\delta$ and $\rightarrow_d$ of Definition 2, we assume that there are two abstract transition relations $\rightarrow_\delta^\alpha$ and $\rightarrow_d^\alpha$ that relate abstract states and constitute *a simulation* of $\rightarrow_\delta$ and $\rightarrow_d$ in the following sense:

1. if $(loc, v_1) \rightarrow_\delta (loc, v_2)$ and $\langle loc, P_1 \rangle$ is an abstraction of (loc, v_1), then there exists an abstract state $\langle loc, P_2 \rangle$ such that $\langle loc, P_1 \rangle \rightarrow_\delta^\alpha \langle loc, P_2 \rangle$, and $\langle loc, P_2 \rangle$ is an abstraction of (loc, v_2).

2. if $(loc_1, v_1) \rightarrow_d (loc_2, v_2)$ and $\langle loc_1, P_1 \rangle$ is an abstraction of (loc_1, v_1), then there exists an abstract state $\langle loc_2, P_2 \rangle$ such that $\langle loc_1, P_1 \rangle \rightarrow_d^\alpha \langle loc_2, P_2 \rangle$, and $\langle loc_2, P_2 \rangle$ is an abstraction of (loc_2, v_2).

Observe that relations $\rightarrow_\delta^\alpha$ and $\rightarrow_d^\alpha$ are generic in the sense that they are only required to simulate the concrete transition relations. We now define abstract relations on extended abstract states:

1. if $\langle loc, P_1\rangle \rightarrow_{\delta}^{\alpha} \langle loc, P_2\rangle$, then for all P such that $P_1 \subseteq P$, we define the abstract continuous transition relation on extended states

$$\langle loc, P_1, P\rangle \rightarrow_{\delta}^{\alpha} \langle loc, P_2, pol(P \cup P_2)\rangle$$

where, given $R \subseteq \mathbb{R}^n$, $pol(R)$ represents the smallest convex polyhedron in $\mathbb{R}^n$ containing R.

2. if $\langle loc_1, P_1\rangle \rightarrow_{d}^{\alpha} \langle loc_2, P_2\rangle$, then for all P such that $P_1 \subseteq P$, we define the abstract discrete transition relation on extended states

$$\langle loc_1, P_1, P\rangle \overset{P}{\rightarrow}_{d}^{\alpha} \langle loc_2, P_2, P_2\rangle$$

Note that $\rightarrow_d^{\alpha}$ is labeled by polyhedron P, which is the third component of its source extended abstract state. We do this to record, in the transition label, the polyhedron accumulated by the preceding abstract continuous transitions at loc.

We can now redefine the notion of trajectory given in Section 2.2 considering (extended) abstract states. Given a convex polyhedron $P_0 \subseteq [[Init(loc_0)]]$, an *extended abstract trajectory* is a sequence of extended abstract states obtained by applying the abstract transitions described above from the initial state $\langle loc_0, P_0, P_0\rangle$. Thus, an extended abstract trajectory could be as follows:

$$\langle loc_0, P_0^0, P_0^0\rangle \rightarrow_{\delta_0^0}^{\alpha} \langle loc_0, P_1^0, A_1^0\rangle \rightarrow_{\delta_1^0}^{\alpha} \cdots \rightarrow_{\delta_{i_0-1}^0}^{\alpha} \langle loc_0, P_{i_0}^0, A_{i_0}^0\rangle \overset{A_{i_0}^0}{\rightarrow}_{d}^{\alpha} \langle loc_1, P_0^1, P_0^1\rangle \cdots$$

where $A_1^0 = pol(P_0^0 \cup P_1^0)$, $A_2^0 = pol(A_1^0 \cup P_2^0), \cdots$ are the accumulated polyhedra obtained during the continuous evolution at a location.

Observe that with this new notion of trajectory which uses polyhedra instead of points, we can jointly describe a non-enumerable set of hybrid trajectories. In addition, by construction, if we fix the location loc_j, then the sets $P_0^j, A_1^j, \cdots, A_{i_j}^j$ corresponding to the continuous transitions in loc_j satisfy the chain of inclusions: $P_0^j \subseteq A_1^j \subseteq \cdots \subseteq A_{i_j}^j$. Thus, the last polyhedron $A_{i_j}^j$ contains all points visited during the continuous abstract transitions executed at location loc_j.

The proposition below captures the simulation relation between trajectories and abstract trajectories:

Proposition 1 *Given a trajectory of a hybrid system* $traj = (loc_0, v_0) \rightarrow_{\lambda_0} (loc_1, v_1) \rightarrow_{\lambda_1} \cdots$, *there exists an extended abstract trajectory* $\alpha(traj) = \langle loc_0, P_0, P_0'\rangle \rightarrow_{\lambda_0}^{\alpha} \langle loc_1, P_1, P_1'\rangle \rightarrow_{\lambda_1}^{\alpha} \cdots$ *such that for all* $i \geq 0$. $\langle loc_i, P_i, P_i'\rangle$ *is an abstraction of* $\langle loc_i, v_i\rangle$.

Namely, each trajectory $traj$ is simulated step by step by an extended abstract trajectory $\alpha(traj)$. The next definition compacts each abstract extended trajectory into an abstract trajectory which only uses the discrete abstract transition relation. This transformation enables the proof of the desired simulation relation between trajectories and abstract discrete trajectories.

Definition 4 (Abstract trace) Consider an *extended abstract trajectory*

$$\alpha(traj) = \langle loc_0, P_0^0, P_0^0\rangle \rightarrow^{\alpha}_{\delta_0^0} \cdots \rightarrow^{\alpha}_{\delta^0_{i_0-1}} \langle loc_0, P_{i_0}^0, A_{i_0}^0\rangle \rightarrow^{A^0_{i_0}}_{d} \langle loc_1, P_0^1, P_0^1\rangle \rightarrow^{\alpha}_{\delta_0^1} \cdots \rightarrow^{\alpha}_{\delta^1_{i_1-1}} \langle loc_1, P_{i_1}^1, A_{i_1}^1\rangle \rightarrow^{A^1_{i_1}}_{d} \cdots$$

then the *abstract trace* $\overline{\alpha(traj)}$ determined by $\alpha(traj)$ is defined as

$$\overline{\alpha(traj)} = \langle loc_0, P_0^0\rangle \xrightarrow{A^0_{i_0}}_{d} \langle loc_1, P_0^1\rangle \xrightarrow{A^1_{i_1}}_{d} \cdots$$

Observe that in the previous definition, the abstract trace $\overline{\alpha(traj)}$ constructed from $\alpha(traj)$ is a sequence of states similar to those determined by labeled transition systems. Each label of $\overline{\alpha(traj)}$ contains the polyhedron $A_{i_j}^j$ accumulated during the continuous transitions at location loc_j which have been abstracted. In other words, we are collapsing the continuous transitions at each location into a single abstract state, encoding in the label of the following discrete transition the accumulated result of executing the continuous part:

$$\underbrace{\langle loc_0, P_0^0, P_0^0\rangle \rightarrow^{\alpha}_{\delta_0^0} \cdots \rightarrow^{\alpha}_{\delta^0_{i_0-1}} \langle loc_0, P_{i_0}^0, A_{i_0}^0\rangle}_{} \rightarrow^{A^0_{i_0}}_{d} \underbrace{\langle loc_1, P_0^1, P_0^1\rangle \rightarrow^{\alpha}_{\delta_0^1} \cdots \rightarrow^{\alpha}_{\delta^1_{i_1-1}} \langle loc_1, P_{i_1}^1, A_{i_1}^1\rangle}_{} \rightarrow^{A^1_{i_1}}_{d} \cdots$$

$$\langle loc_0, P_0^0\rangle \rightarrow^{A^0_{i_0}}_{d} \langle loc_1, P_0^1\rangle \rightarrow^{A^1_{i_1}}_{d} \cdots$$

Using Definition 4 and Proposition 1, we can formalize the simulation of the hybrid trajectories by a discrete sequence of abstract states, as follows:

Theorem 1 *Given a trajectory of a hybrid system*

$$traj = (loc_0, v_0^0) \rightarrow_{\delta_0^0} \cdots \rightarrow_{\delta^0_{i_0-1}} (loc_0, v_{i_0}^0) \rightarrow_d (loc_1, v_0^1) \rightarrow_{\delta_0^1} \cdots$$

there exists a sequence of abstract states

$$\overline{\alpha(traj)} = \langle loc_0, P_0\rangle \rightarrow^{A^0}_{d} \langle loc_1, P_1\rangle \rightarrow^{A^1}_{d} \cdots$$

such that for all $k \geq 0, 0 \leq l \leq i_k$. $v_l^k \in A^k$.

Consequently, each possible trajectory *traj* of a rectangular hybrid system is simulated by a path $\overline{\alpha(traj)}$ that uses polyhedra as abstractions of the real values.

3.1 Analyzing Temporal Properties.

Given a hybrid automaton H, we denote with $Sem(H)$ the set of all possible trajectories produced by H. Let $Sem^{\alpha}(H)$ denote the set of the traces of abstract states $\{\overline{\alpha(traj)} | traj \in Sem(H)\}$ constructed as above from the trajectories in $Sem(H)$.

In order to analyze properties of the trajectories in $Sem(H)$, we analyze the properties satisfied by the paths in $Sem^{\alpha}(H)$. However, an abstraction can suffer from loss of information, which

means that the satisfaction of properties on $Sem^{\alpha}(H)$ cannot always be transferred to $Sem(H)$. The rest of the section is devoted to discussing this aspect.

Let us assume that $Prop$ is a set of atomic propositions to be evaluated on hybrid states of automaton H. Thus, given a state (loc, v), and a proposition $p \in Prop$, we write $(loc, v) \models p$ when (loc, v) satisfies p. Now consider an abstract state $\langle loc, P\rangle$. We have two dual ways of defining when $\langle loc, P\rangle$ satisfies a proposition p:

1. We say that $\langle loc, P\rangle$ *under*-satisfies p, and denote this with $\langle loc, P\rangle \models_u^{\alpha} p$ iff $\forall v \in P.$ $(loc, v) \models p$.
2. We say that $\langle loc, P\rangle$ *over*-satisfies p, and denote this with $\langle loc, P\rangle \models_o^{\alpha} p$ iff $\exists v \in P.$ $(loc, v) \models p$.

Thus, $\langle loc, P\rangle \models_u^{\alpha} p$ means that all hybrid states abstracted by $\langle loc, P\rangle$ satisfy p. However, $\langle loc, P\rangle \models_o^{\alpha} p$ means that at least one hybrid state abstracted by $\langle loc, P\rangle$ satisfies p. Clearly, $\langle loc, P\rangle \models_u^{\alpha} p \Rightarrow \langle loc, P\rangle \models_o^{\alpha} p$.

Operators $\models$, $\models_u^{\alpha}$ and $\models_o^{\alpha}$ can be inductively extended to (abstract) trajectories and LTL formulas as usual. Given a trajectory $traj \in Sem(H)$, and an LTL formula f, we write $traj \models f$ iff $traj$ satisfies f, (according to the semantics of LTL). Similarly, we write $H \models \forall f$ iff $\forall traj \in Sem(H).traj \models f$, and $H \models \exists f$ iff $\exists traj \in Sem(H).traj \models f$. Similar definitions can be done with LTL formulas, $\models_u^{\alpha}$ and $\models_o^{\alpha}$ and abstract trajectories.

The following proposition, proved in [GMP02], establishes the results of property preservation between the abstract and hybrid trajectories.

Proposition 2 *Given a hybrid automata H, and a temporal* LTL *formula f:*

1. *if* $Sem^{\alpha}(H) \models_u^{\alpha} \forall f$ *then* $Sem(H) \models \forall f$
2. *if* $Sem^{\alpha}(H) \not\models_o^{\alpha} \exists f$ *then* $Sem(H) \not\models \exists f$

In other words, abstraction is useful to prove universal formulae (if they are *under*-approximated) or to refute existential properties (if they are *over*-approximated). Otherwise, the counterexample obtained when trying to prove the property must be analyzed to check whether it corresponds to a real behavior of the automaton.

However, there are some formulae for which under and over satisfaction of properties coincide. For instance, assume that p is an atomic proposition describing property "*automaton is at location l*". It is trivial to prove that $\langle loc, P\rangle \models_u^{\alpha} p \iff \langle loc, P\rangle \models_o^{\alpha} p$, which means that we may prove or refute properties that only contain atomic propositions like p. In [GMMP04], there is a general introduction to temporal logic analysis on abstract systems.

3.2 Analyzing Deadlocks

We briefly discuss the case in which the abstract transition relation $\rightarrow_{\delta}^{\alpha}$ and $\rightarrow_{d}^{\alpha}$ are not only a simulation of the original $\rightarrow_{\delta}$ and $\rightarrow_{d}$, but are also bi-simulations in the usual sense.

Assume that transition relations $\rightarrow_{\delta}^{\alpha}$ and $\rightarrow_{d}^{\alpha}$ bi-simulate relations $\rightarrow_{\delta}$ and $\rightarrow_{d}$, i.e., in addition to being similar to the original transition relations, they also satisfy the following conditions.

1. if $\langle loc, P_1\rangle \rightarrow_{\delta}^{\alpha} \langle loc, P_2\rangle$ and abstract state $\langle loc, P_1\rangle$ is an abstraction of (loc, v_1), there exists a state (loc, v_2) such that $(loc, v_1) \rightarrow_{\delta} (loc, v_2)$, and $\langle loc, P_2\rangle$ is an abstraction of (loc, v_2).

2. if $\langle loc_1, P_1\rangle \rightarrow_{d}^{\alpha} \langle loc_2, P_2\rangle$ and abstract state $\langle loc_1, P_1\rangle$ is an abstraction of (loc_1, v_1), there exists a state (loc_2, v_2) such that $(loc_1, v_1) \rightarrow_{d} (loc_2, v_2)$, and $\langle loc_2, P_2\rangle$ is an abstraction of (loc_2, v_2).

Definition 5 (Blocked sequence) Given a general linear transition system (LTS) $(\Sigma', \rightarrow)$, and a sequence of states $seq = s_0 \rightarrow s_1 \rightarrow \cdots$, we say that *seq* is a *blocked sequence* if there exists a non end state s_i in the sequence, such that $s_i \not\rightarrow$.

Observe that this definition applies to any LTS, and consequently it also refers to abstract/concrete trajectories of a hybrid system.

Proposition 3 *Let H be a hybrid system, and assume that $Sem(H)$ and $Sem^{\alpha}(H)$ are, respectively, the set of trajectories and abstract paths determined by the original transition relations $\rightarrow_{\delta}$ and $\rightarrow_{d}$, and two abstract relations $\rightarrow_{\delta}^{\alpha}$ and $\rightarrow_{d}^{\alpha}$ which constitute bi-simulations as defined above. Then, if $\forall \overline{\alpha(traj)} \in Sem^{\alpha}(H), \overline{\alpha(traj)}$ does not block, there exists a trajectory $traj \in Sem(H)$ which does not block either.*

4 Extending JPF

To extend JPF without modifying the *jpf-core*, it is necessary to develop a new *jpf-project* which includes the model classes, the native peers, etc. related with the new extension. All *jpf-projects* are organized following the same source folder tree. It is important to know these folders because classes are executed by JPF or by the JVM depending on their location. The main folders are:

- src/main: contains classes executed by the JVM (listeners, choice generators, etc).
- src/peers: contains the library classes executed by the JVM (MJI native peers).
- src/classes: contains the library classes executed by JPF.
- src/example: contains the models and the configuration file of the system under analysis.

In this section, we first present how to model an IRHA using Java. Then, we briefly describe the implementation of the JPF extension for hybrid systems. We introduce the main classes developed, and present some fragments of code to illustrate how to use them.

4.1 Modeling Hybrid Automata

Modeling hybrid systems in JPF do not require extending the JPF input language. Since Java is an object oriented program, an automaton will be a object of a special class. The same idea is used to model other components such as continuous variables or locations. All these new classes

```
1 public class FanController_dtr extends RAutomaton{
2   private ContVar temp, clock;
3   private DiscTrans ton_off, toff_on;
4   private HybridState on, of;
5
6   public FanController(int id){
7     super(id);
8     temp= new ContVar("[10,15]","[1,1]","temp",id);
9     clock= new ContVar("[0,0]","[1,1]","clock",id);
10    off= new HybridState("off", "temp<=35", true,id);
11    on= new HybridState("on", "temp>=15", false,id);
12    ton_off= new DiscTrans("on", "off", "temp<=25", "clock={,[0,0]}", id);
13    toff_on= new DiscTrans("off", "on", "temp>=30 && clock>=10", "clock
          ={,[0,0]}", id);
14  }
15 }
```

Figure 4: Java model of the fan controller

are executed by JPF and constitute a library for later modeling specific hybrid systems. This means that the four classes described in this section are located in the folder src/classes.

The class *RAutomaton* describes a rectangular hybrid automaton. This class extends from the class Thread implemented in JPF. In consequence, we do not have to implement mechanisms for interleaving the discrete transitions of automaton. The class RAutomaton does not include concrete locations or continuous variables. The description of an automaton, such as the fan controller, has to extend from RAutomaton, and declare the specific variables and locations as member variables.

The class *HybridState* defines an automaton location. Each HybridState object includes the name, the invariant and a flag that indicates whether the location is the initial one. The class *ContVar* models a continuous variable. Each ContVar object stores the value and the flow equation of the variable in the initial location. In addition, the object has an identifier, which corresponds with the dimension that represents the variable in the polyhedra. Class *DiscTrans* defines discrete transitions between locations. It is characterized by the names of the source and target location, the guard condition that enables the transition, and the updates that have to be performed when the transition is fired. The constructor of all these classes also includes the automaton identifier.

Using these five classes, any IRHA can be depicted. For instance, Figure 4 shows the code of the fan controller introduced in Section 2.2.

4.2 Definition of the External Structures

As presented in Section 3, continuous trajectories of hybrid systems can be analyzed as discrete traces applying some abstraction to the continuous behavior of the automata. In this section, we describe some basic structures used to store the abstraction of the continuous variables when analyzing a hybrid system with JPF. These structures are external and transparent to JPF in the sense that the class that contains its definition is executed by the host JVM, and not by JPF. Thus, the data store in the structures is not tracked by JPF.

- **Global Location:** This variable, denoted as *gloc*, stores the location of all concurrent automata of the system. Assuming that we have m concurrent automata, the system state is represented as $s_e = \langle (loc_1, sv_1), \cdots, (loc_m, sv_m), P_G \rangle$, $gloc$ stores the product location $gloc = (loc_1, \cdots, loc_m)$. Global location is used as a discrete reference of the state of the product automaton. Thus, when its value changes we force the generation of a new JPF transition that evaluates whether the transition generates a new or visited state.

- **Polyhedra Pool:** This structure stores all the convex polyhedra visited during the analysis, irrespective of the state where they have been reached. That is to say, it stores the successive polyhedra, which are abstractions of the continuous variables, computed during the analysis. These polyhedra were denoted as $A^0, A^1 \cdots$ in Theorem 1. When a new polyhedron is calculated, if it is not contained in any polyhedra previously found, it is stored in the pool, and we associate it with a fresh identifier.

- **Global Location Table:** This data structure stores the polyhedra visited in each location of the product automaton. Each table component corresponds to a *gloc* value along with a list of reached polyhedra. To reduce the memory consumption, this table only stores the polyhedra identifiers.

  The global location table is used as follows to know whether an extended state has been previously visited. Assuming that the discrete part of the state coincides, and that the polyhedron just calculated is contained in a polyhedron indexed in the corresponding component of the *global location table*, we may conclude that the system state has already been visited, and the model checker may backtrack.

4.3 Including the External Structures

We now explain how to include the structures defined in Section 4.2 in our jpf-project. We also report on the analysis carried out by JPF and the polyhedra computation using PPL.

Since JPF cannot run native methods, such as calls to PPL methods, we have to delegate the execution of these methods to the host JVM. To this end, we use the MJI interface, that intercepts the calls to native methods, and the native peer classes, that implement these methods. In addition, native peer classes include variables and structures which are hidden from JPF.

To centralize the management of the structures presented in Section 4.2, we have developed an extra model class, called *Analyzer* and an associated native peer. The *Analyzer* model class has two member variables, the current global location and the identifier of the current polyhedra, which are part of JPF state. The rest of the structures are defined in the Analyzer's native peer, namely, they are hidden from JPF and store objects of unsupported types by JPF. We use Java collections to define these structures, in this way the development is easier because we do not need to efficiently implement the methods to include or extract data from the structures.

We have also developed native peers for all the classes presented Section 4.1. Each native peer implements the constructor of the respective model class, to store the relevant data inside the structures presented in Section 4.2.

```
1 public class TestFanController {
2
3   public static void main(String[] args) {
4     System.out.println("Starting TestFanController..");
5     Analyzer a= new Analyzer(0);
6     FanController f1= new FanController(1);
7     f1.start();
8   }
9 }
```

Figure 5: Test class for fan controller

4.4 Analyzing systems with JPF extended for hybrid systems

To analyze a system with JPF, we have to define a *main* method in one of the model classes. We recommend using an extra class that contains the whole model. Figure 5 shows this class for the fan controller example. Note that, apart from a FanController, we need an Analyzer, which is in charge of initializing the structures and loading the PPL library at execution time.

Finally, it is necessary to define a *jpf-property* file. Although, there are many aspects that can be configured, the most important one is the target class, that contains the main method and the properties to be checked. This is the file used to invoke JPF.

5 Conclusions

In this paper, we have presented a work in progress to extend the JPF model checker for hybrid system analysis. In previous work, we proposed a methodology that extends explicit discrete model checkers, and we applied it to SPIN. Now, we prove that the approach is sufficiently generic to be applied to JPF too. We also formalize the methodology, proving its soundness.

Although the prototype is still not completely operative, the current results are promising. JPF provides the necessary mechanisms to be extended in a non-intrusive way, allowing the use of external libraries, such as PPL, to perform the abstraction of the continuous behavior of a hybrid system.

Future work will focus on finishing the implementation of the prototype and analyzing some relevant case studies. One of the open issues is to implement the synchronous communication of automata in JPF. In addition, we would like to compare the performance of the tool with other existing tools for hybrid systems.

Acknowledgements: This work has been funded by Spanish projects TIN 2008-05932 and WITLE2 IDI-20090382, by the Andalusian project P07-TIC3131 and ERDF from the European Commission.

Bibliography

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P. Ho, X. Nicollin, A. Olivero, J. Sifakis, S. Yovine. The algorithmic analysis of hybrid systems. *Theor. Comput. Sci.* 138:3–34, 1995.

[AD91] R. Alur, D. Dill. The Theory of Timed Automata. In *Procs. of REX Workshop*. Pp. 45–73. 1991.

[BBB+00] A. Balluchi, L. Benvenuti, M. D. Benedetto, C. Pinello, A. Sangiovanni-Vincentelli. Automotive Engine Control and Hybrid Systems: Challenges and Opportunities. *Procs. of the IEEE. Special Issue on Hybrid Systems* 7:888–912, 2000.

[BHZ08] R. Bagnara, P. Hill, E. Zaffanella. The Parma Polyhedra Library: Toward a Complete Set of Numerical Abstractions for the Analysis and Verification of Hardware and Software Systems. *Science of Computer Programming* 72(1–2):3–21, 2008.

[BMP99] O. Bournez, O. Maler, A. Pnueli. Orthogonal Polyhedra: Representation and Computation. In *Procs. of HSCC'99*. Pp. 46–60. Springer-Verlag, 1999.

[CK98] A. Chutinan, B. Krogh. Computing polyhedral approximations to flow pipes for dynamic systems. In *Procs. of IEEE CDC*. IEEE press, 1998.

[Feh99] A. Fehnker. Scheduling a Steel Plant with Timed Automata. In *Procs. of RTCSA '99*. Pp. 280–286. IEEE Computer Society, 1999.

[Fre05] G. Frehse. *Compositional verification of hybrid systems using simulation relations*. IPA dissertation series. 2005.

[GMMP04] M.-d.-M. Gallardo, J. Martínez, P. Merino, E. Pimentel. aSPIN: A tool for abstract model checking. *STTT* 5(2-3):165–184, 2004.

[GMP02] M.-d.-M. Gallardo, P. Merino, E. Pimentel. Refinement of LTL Formulas for Abstract Model Checking. In *Procs. of SAS*. Pp. 395–410. 2002.

[GP10] M.-d.-M. Gallardo, L. Panizo. An approach to verify hybrid systems with SPIN. In *Procs. of PROLE 2010 (Sistedes)*. 2010.

[Hen96] T. Henzinger. The theory of Hybrid Automata. In *Procs. of LICS '96*. Pp. 278–292. IEEE Computer Society, 1996.

[HHW98] T. Henzinger, P.-H. Ho, H. Wong-toi. Algorithmic Analysis of Nonlinear Hybrid Systems. *IEEE Transactions on Automatic Control* 43:540–554, 1998.

[HK99] T. A. Henzinger, P. Kopke. Discrete-Time Control for Rectangular Hybrid Automata. *Theor. Comput. Sci.* 221(1-2):369–392, 1999.

[HKPV98] T. Henzinger, P. Kopke, A. Puri, P. Varaiya. What's decidable about hybrid automata? *J. Comput. Syst. Sci.* 57:94–124, 1998.

[Ho95] P.-H. Ho. *Automatic analysis of hybrid systems*. PhD thesis, Ithaca, NY, USA, 1995.

[Jea02] B. Jeannet. Convex polyhedra library, Documentation of the New Polka library. 2002. Available at: http://www.irisa.fr/prive/Bertrand.Jeannet/newpolka.html.

[jpf12] Java Path Finder. May 2012. http://babelfish.arc.nasa.gov/trac/jpf/wiki/WikiStart

[KMP+95] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, D. Wonnacott. The Omega Library interface guide. Technical report, University of Maryland at College Park, 1995.

[Kop96] P. Kopke. The Theory of Rectangular Hybrid Automata. Technical report, Cornell University, Ithaca, NY, USA, 1996.

[LCW+02] V. Loechner, P. Clauss, D. Wilde, B. Meister, R. Seghir, G. Bitran, J. Léger. PolyLib: A Library for doing Symbolic Polyhedra Operations. In *Procs. of ISSAC2002*. 2002.

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): Actas de las "XII Jornadas sobre Programación y Lenguajes (PROLE'2012), Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

Charla Invitada

Strategy-Driven Graph Transformations in PORGY

Maribel Fernández

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): Actas de las "XII Jornadas sobre Programación y Lenguajes (PROLE'2012), Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

Strategy-Driven Graph Transformations in PORGY

Maribel Fernández

King's College London, Strand, London WC2R 2LS, UK
Maribel.Fernandez@kcl.ac.uk

PORGY [2] is a visual and interactive tool designed to facilitate the specification and analysis of complex systems. PORGY uses graphs to represent the system modelled, and graph rewrite rules guided by strategies to describe the dynamics of the system.

Graphical languages are widely used for describing complex structures in a visual and intuitive way in a variety of domains, such as software modelling (e.g., UML diagrams), representation of proofs (e.g., proof nets), microprocessor design, XML documents, communication networks, and biological systems, amongst others. Graph transformations (or graph rewriting) are used to define the behaviour of the system modelled. From a theoretical point of view, graph rewriting has solid logic, algebraic and categorical foundations [3, 9], and from a practical point of view, it has many applications in specification, programming, and simulation [4, 5].

When the graphs are large or growing via transformations, or when the number of transformation rules is important, being able to directly interact with the rewriting system becomes crucial to understand the changes in the graph structure. From a naïve point of view, the output of a graph rewriting system is a dynamic graph: a sequence of graphs obtained through a series of modifications (addition/deletion of nodes/edges). However, the study of a rewriting system is actually much more complex. Reasoning about the system's properties actually involves testing various rewriting scenarios, backtracking to a previously computed graph, possibly updating rules, etc. PORGY, developed in collaboration with INRIA Bordeaux-Sud Ouest (http://gravite.labri.fr/?Projects_%2F_Grants:Porgy:Download), addresses these issues.

Our approach is based on the use of port graphs and port graph rewriting rules [1]. Port-graphs are a specific class of labelled graphs introduced as an abstract representation of proteins, and used to model biochemical interactions and autonomous systems. We illustrate this concept by using port graph transformations to model biochemical systems (biochemical interactions that take part in the regulation of cell proliferation and transformation) and interaction nets. These case studies illustrate the need for highly dynamic graphs, and highlight interesting challenges for graph visualisation. PORGY provides support for the initial task of defining a set of graph rewriting rules, and the graph representing the initial state of the system (the "initial model" in PORGY's terminology), using a visual editor.

Other crucial issues concern when and where rules are applied. To address this problem, PORGY provides a strategy language to constrain the rewriting derivations, generalising the control structures used in graph-based tools such as PROGRES [10] and GP [8], and rewrite-based programming languages such as Stratego and ELAN. In particular, the strategy language includes control structures that facilitate the implementation of graph traversal algorithms, thanks to the explicit definition of "positions" in a graph, where rules can be applied (we refer the reader

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): PROLE'2012, pp. 253-254, ISBN:978-84-15487-27-2.
Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

to [7] for examples of graph programs in PORGY, and to [6] for the formal semantics of the strategy language).

Rewriting derivations can also be visualised, and used in an interactive way, using PORGY's interface. Designing a graph transformation system is often a complex task, and the analysis and debugging of the system involves exploring how rules operate on graphs, analysing sequences of transformations, backtracking and changing earlier decisions. For this purpose, PORGY's visual environment offers a view on the rewriting history and ways to select time points in the history where to backtrack.

Bibliography

[1] Oana Andrei (2008): *A Rewriting Calculus for Graphs: Applications to Biology and Autonomous Systems*. Ph.D. thesis, Institut National Polytechnique de Lorraine. Available at http://tel.archives-ouvertes.fr/tel-00337558/fr/.

[2] Oana Andrei, Maribel Fernández, Hélène Kirchner, Guy Melançon, Olivier Namet & Bruno Pinaud (2011): *PORGY: Strategy-Driven Interactive Transformation of Graphs*. *Proceedings of the 6th International Workshop on Computing with Terms and Graphs, TERMGRAPH 2011, EPTCS*.

[3] Bruno Courcelle (1990): *Graph Rewriting: An Algebraic and Logic Approach*. In J. van Leeuwen, editor: *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. Elsevier Science Publishers and MIT Press, pp. 193–242.

[4] Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski & Grzegorz Rozenberg, editors (1997): *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 2: Applications, Languages, and Tools*. World Scientific.

[5] Hartmut Ehrig, Hans-Jörg Kreowski, Ugo Montanari & Grzegorz Rozenberg, editors (1997): *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 3: Concurrency, Parallelism, and Distribution*. World Scientific.

[6] Maribel Fernández, Hélène Kirchner & Olivier Namet (2012): *A strategy language for graph rewriting*. To appear in Proceedings of LOPSTR 2011.

[7] Maribel Fernández & Olivier Namet (2010): *Strategic Programming on Graph Rewriting Systems*. In: *Proc. of the* 1^{st} *International Workshop on Strategies in Rewriting, Proving, and Programming*.

[8] Detlef Plump (2009): *The Graph Programming Language GP*. In Symeon Bozapalidis & George Rahonis, editors: *CAI. Lecture Notes in Computer Science* 5725, Springer, pp. 99–122.

[9] Grzegorz Rozenberg, editor (1997): *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific.

[10] Andy Schürr, Andreas J. Winter & Albert Zündorf (1997): *The PROGRES Approach: Language and Environment*. In: *[4]*. World Scientific, pp. 479–546.

Taller de Programación Funcional

TPF'2012

Mateu Villavert

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): Actas de las "XII Jornadas sobre Programación y Lenguajes (PROLE'2012), Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

Sesión 1

Ponencias invitadas

Sesión 1: Ponencias invitadas

Jose Iborra. *Eden: An F#/WPF framework for building GUI tools*.

Pablo Nogueira. *Los cálculos lambda y lambda-value y sus estrategias de reducción*.

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): Actas de las "XII Jornadas sobre Programación y Lenguajes (PROLE'2012), Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

Eden: An F#/WPF framework for building GUI tools (Charla invitada)

José Iborra López

Credit Suisse

Abstract: Our group within Credit Suisse is responsible for developing quantitative models used to value financial products within the Securities Division of the bank. One aspect of this role is to deliver tools based on those models to trading and sales staff, which they can use to quickly price proposed transactions and perform other analysis of market conditions. Historically these tools have been delivered as Excel spreadsheets. WPF (Windows Presentation Foundation) is a GUI framework which encourages architectural separation between the layout of the user interface itself (the "View") and the underlying interactions and calculations (the "ViewModel" and "Model"). We have built a framework for developing tools in WPF that makes use of a graph-based calculation engine for implementing ViewModels and Models in F#. The engine is built on F# asynchronous workflows and provides a standard set of features to our tools. In this talk I'll discuss the implementation of this calculation engine, including various steps in its evolution that led up to our use of asynchronous workflows. I'll also talk about how well F# and asynchronous workflows have worked for us, and briefly discuss some of the challenges of integrating F# and WPF.

Keywords: FP, F#, GUI.

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): PROLE'2012, pp. 259, ISBN:978-84-15487-27-2.
Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): Actas de las "XII Jornadas sobre Programación y Lenguajes (PROLE'2012), Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

Los cálculos lambda y lambda-value y sus estrategias de reducción (Seminario)

Pablo Nogueira[*]

pablo@babel.ls.fi.upm.es
Universidad Politécnica de Madrid
Spain

Abstract: El cálculo lambda es el núcleo de los lenguajes funcionales. El cálculo lambda puro es la versión clásica más básica. Menos conocidos son el cálculo lambda-value de Plotkin y la generalización de ámbos, el cálculo lambda-paramétrico de Paolini y Ronchi della Rocca. En este seminario me gustaría dar a conocer dichos cálculos, los conceptos fundamentales (conversión, reducción, estrategias de reducción small y big-step, compleción, estandarización, solvability, etc.) y por qué son útiles para el estudio de los lenguajes de programación funcional.

Keywords: Cálculo lambda, estrategias de reducción.

[*] Trabajo con financiación parcial de proyectos MINECO TIN2009-14599-C03-00 (DESAFIOS10) y CAM P2009/TIC-1465 (PROMETIDOS).

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): PROLE'2012, pp. 261, ISBN:978-84-15487-27-2.
Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): Actas de las "XII Jornadas sobre Programación y Lenguajes (PROLE'2012), Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

Sesión 2

Artículos aceptados

Sesión 2: Artículos

Alvaro Garcia-Perez and Pablo Nogueira. *Enfoque Normal y Enfoque Spine para Reducción en el Cálculo Lambda Puro*.

David Insa, Josep Silva, Salvador Tamarit and César Tomás. *Fragmentación de Programas con Emparejamiento de Patrones*.

Macías López Iglesias, Laura M. Castro and David Cabrero. *Sistema funcional distribuido de publicidad para iDTV*.

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): Actas de las "XII Jornadas sobre Programación y Lenguajes (PROLE'2012), Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

Enfoque Normal y Enfoque *Spine* para Reducción en el Cálculo Lambda Puro

Álvaro García-Pérez[1*] **and Pablo Nogueira**[2**]

[1] agarcia@babel.ls.fi.upm.es
IMDEA Software Institute
Universidad Politécnica de Madrid, Spain

[2] pablo@babel.ls.fi.upm.es
Universidad Politécnica de Madrid, Spain

Abstract: El enfoque tradicional para la reducción en el cálculo lambda puro [Bar84] considera estrategias estándar [CF58], que nunca contraen un *redex* que esté a la izquierda del contracto de algún otro redex previamente contraído. La estrategia *normal order* representa dicho enfoque—al que nos referimos como *enfoque normal*—, contrayendo el redex situado más a la izquierda hasta que se alcanza una forma normal. Normal order elimina todos los *redices* presentes en el término, incluso aquellos que se encuentran en el cuerpo de una abstracción o a la derecha de una variable libre. Normal order es una estrategia completa, ya que encuentra la forma normal de un término si ésta existe, o diverge en caso contrario. Normal order es una estrategia *híbrida*, donde su definición *big-step* utiliza a la estrategia *call-by-name* de forma subsidiaria. A diferencia de normal order, call-by-name no contrae ningún redex que se encuentre bajo lambda ni a la derecha de una variable libre [Abr90], produciendo formas normales *weak-head*. Normal order utiliza call-by-name para localizar el redex más exterior, contrayendo primero el redex $(\lambda x.B)N$ en vez de contraer prematuramente algún redex en B.

Existe un enfoque alternativo que recorre la espina del término (penetrando en el cuerpo de las abstracciones) antes de contraer el redex más exterior. Denominamos a éste *enfoque spine*. Las estrategia *spine order* reduce bajo lambda y devuelve formas normales, utilizando *head spine* como estrategia subsidiaria [Ses02]. Head spine reduce los cuerpos de las abstracciones pero nunca a la derecha de una variable libre, produciendo formas normales *head*. Spine order utiliza un esquema de hibridación similar al de normal order, tomando head spine como subsidiaria. Spine order contraerá primero el redex $(\lambda x.B)N$—donde B es una forma normal *head*—en vez de contraer prematuramente algún redex situado a la derecha de una variable libre dentro de B.

¿Cuáles son las estrategias normal y *spine* en el cálculo lambda *value* [Plo75]? Dichas estrategias deben ser análogas a normal order y spine order, es decir, deben ser completas y deben devolver formas normales, utilizando el enfoque normal o

* Autor con beca CAM CPI/0622/2008.
** Trabajo con financiación parcial de proyectos MINECO TIN2009-14599-C03-00 (DESAFIOS10) y CAM P2009/TIC-1465 (PROMETIDOS).

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): PROLE'2012, pp. 265-266, ISBN:978-84-15487-27-2.
Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

spine respectivamente. La respuesta es *value normal order* y *value spine order* respectivamente. En este trabajo damos una definición precisa de estrategias híbridas, construimos value normal order y value spine order a partir de consideraciones metateóricas y generalizamos el Teorema de Estandarización de Plotkin, demostrando un teorema análogo al Teorema de Reducción *Quasi-Leftmost* [HS08, p.40], sustentando así el enfoque spine en el cálculo lambda value.

Keywords: Cálculo lambda, estrategias de reducción, semántica operacional.

Referencias

[Abr90] S. Abramsky. The Lazy Lambda Calculus. In Turner (ed.), *Research Topics in Functional Programming*. Pp. 65–116. Addison-Welsey, Reading, MA, 1990.

[Bar84] H. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*. North Holland, 1984.

[CF58] H. B. Curry, R. Feys. *Combinatory Logic*. Volume 1. North-Holland, 1958.

[HS08] J. Hindley, J. Seldin. *Lambda-calculus and combinators, an introduction*. Cambridge University Press, 2008.

[Plo75] G. Plotkin. Call-by-name, Call-by-value and the Lambda Calculus. *Theoretical Computer Science* 1:125–159, 1975.

[Ses02] P. Sestoft. Demonstrating Lambda Calculus Reduction. In *The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*. Lecture Notes in Computer Science 2566, pp. 420–435. Springer, 2002.

Fragmentación de Programas con Ajuste de Patrones *

David Insa[1], Josep Silva[2], Salvador Tamarit[3], César Tomás[4]

[1] dinsa@dsic.upv.es, [2] jsilva@dsic.upv.es, [3] stamarit@dsic.upv.es, [4] ctomas@dsic.upv.es
Universitat Politècnica de València
Camino de Vera s/n, E-46022 Valencia, Spain.

Abstract: Una de las técnicas de análisis de programas más extendidas es la fragmentación de programas. Sin embargo, a pesar de que la técnica se utiliza en multitud de áreas (como la depuración, el testeo, la especialización de programas, etc.), su utilización en lenguajes fuera del paradigma imperativo está bastante limitada. La razón fundamental es que los algoritmos actuales ignoran características propias de otros paradigmas distintos al imperativo, tales como la pereza, el orden superior o el ajuste de patrones. En este trabajo proponemos una técnica de fragmentación para abordar de manera eficaz el ajuste de patrones.

Keywords: Fragmentación de Programas, Ajuste de Patrones, Erlang.

1. Introducción

Desde que Mark Weiser acuñó el término fragmentación de programas (del inglés *Program Slicing*), gran cantidad de trabajos y aplicaciones han sido desarrollados en torno a esta técnica de análisis de programas. La finalidad principal de la fragmentación de programas es la de extraer una parte del programa (o *fragmento*) que influye o es influenciada por un determinado punto de interés (o *criterio de fragmentación*) [Wei81, Tip95, Sil12].

La técnica de fragmentación de programas estática ha estado tradicionalmente basada en el uso de estructuras de datos capaces de representar todas las ejecuciones posibles de un programa. Una de las primeras estructuras propuesta fue el *Program Dependence Graph* (PDG) [FOW87], con el que se pueden computar los fragmentos en tiempo lineal con respecto al tamaño del programa. Sin embargo, el PDG proporciona fragmentos imprecisos en programas interprocedurales. El *System Dependence Graph* (SDG) [HRB90] soluciona este problema guardando el contexto de las llamadas a funciones para poder así diferenciarlas. Posteriormente, el *Erlang Dependence Graph* (EDG) [STT12] ha sido propuesto como la primera adaptación del SDG a un lenguaje funcional (Erlang).

Tradicionalmente la técnica de fragmentación ha estado fuertemente ligada al marco de los lenguajes imperativos, siendo pocas las técnicas propuestas en el contexto de los lenguajes funcionales; algunas excepciones son [FRT95, RT96, OSV04, STT12]. Recientemente, en [STT12] se ha propuesto una técnica de fragmentación de programas para el lenguaje Erlang. Sin embargo, la técnica propuesta carece de un tratamiento específico para el ajuste de patrones (característica

* Este trabajo ha sido parcialmente financiado por el *Ministerio de Economía y Competitividad (Secretaría de Estado de Investigación, Desarrollo e Innovación)* con referencia TIN2008-06622-C03-02 y por la *Generalitat Valenciana* con referencia PROMETEO/2011/052. David Insa y Salvador Tamarit han sido parcialmente financiados por el Ministerio de Eduación con becas FPU AP2010-4415 y FPI BES-2009-015019 respectivamente.

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): PROLE'2012, pp. 267-275, ISBN:978-84-15487-27-2.
Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

ausente en los lenguajes imperativos) y por ello pierde precisión al producir fragmentos de programas que hagan uso de él. En este trabajo abordamos ese problema y proponemos una mejora del EDG para mejorar su precisión mediante el tratamiento eficaz del ajuste de patrones.

El ajuste de patrones es uno de los recursos más utilizados a la hora de programar en lenguajes funcionales como Erlang. Se puede encontrar en asignaciones, instrucciones `case` y `receive` o paso de argumentos en llamadas a funciones. Es por esto que aumentar la precisión de los fragmentos cuando se trata este tipo de construcciones implica una mejora en el resultado global. De esta forma, el trabajo que aquí se presenta introduce una nueva definición del EDG en la que se hace un tratamiento mejorado del ajuste de patrones mediante la introducción de tres nuevos tipos de arcos y varias modificaciones del algoritmo de recorrido.

El artículo se estructura de la siguiente forma: La sección 2 muestra brevemente cómo se construye el EDG y describe el algoritmo utilizado para obtener fragmentos a partir del mismo. Las modificaciones introducidas en el EDG y en el algoritmo se muestran en la sección 3. Finalmente, la sección 4 presenta las conclusiones.

2. Erlang Dependence Graph

Tal y como se ha comentado en la sección anterior, el EDG es la primera adaptación del SDG a un lenguaje funcional. Esta adaptación no es trivial, ya que los lenguajes funcionales imponen nuevas dificultades en la definición del SDG y de los algoritmos usados sobre el mismo. Características como la asignación única, la selección de cláusulas mediante ajuste de patrones, las funciones de orden superior y otras construcciones sintácticas normalmente no presentes en lenguajes imperativos, hacen que la definición tradicional del SDG no sea útil para Erlang. Al igual que el SDG, el EDG se basa en la representación mediante un grafo de los distintos componentes de un programa, es decir, se construye composicionalmente a partir de estructuras predefinidas para cada componente. De esta forma, los nodos representan dichas componentes y los arcos representan dependencias entre las mismas. Tanto nodos como arcos pueden ser de distintos tipos, que explicaremos brevemente a continuación. Una descripción detallada de los distintos componentes del grafo y su construcción se puede encontrar en [STT12].

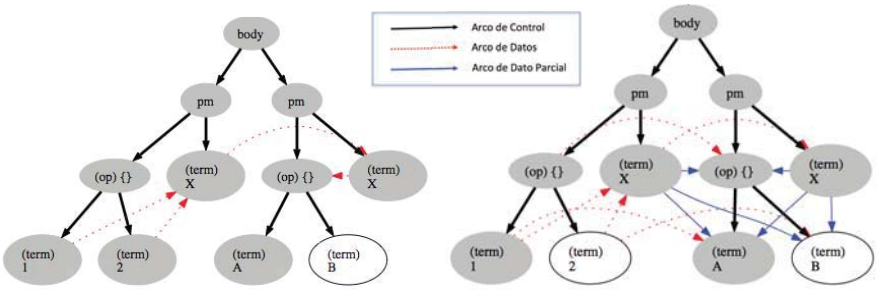
2.1. Representación de estructuras sintácticas en el EDG

Todas las estructuras sintácticas de un programa se representan mediante nodos en el EDG. Adicionalmente, el EDG incluye nodos especiales utilizados para organizar los distintos componentes sintácticos y para representar casos particulares que facilitan el recorrido del mismo. Cada una de las estructuras sintácticas de un programa se representa en el EDG de la siguiente manera:

Variables y literales: Representados mediante un único nodo (de tipo *term*).

Operaciones, Tuplas y Listas: Los tres se representan con un nodo (de tipo *op*) al cual se enlazan todos los grafos de sus operandos o elementos.

Ajustes de patrones y patrones compuestos: El ajuste de patrones enlaza un patrón a una expresión, mientras que los patrones compuestos enlazan dos patrones. En ambos casos se representan con un nodo (de tipo *pm*) unido a los grafos de sus componentes.



(a) EDG original

(b) EDG con Arcos de Dato Parcial de Entrada y nuevos arcos de datos

Figura 1: EDG original (a) y mejorado (b) asociados al código del Ejemplo 1.

Bloques *begin–end*: Se representan con un nodo (de tipo *block*) enlazado con los grafos asociados a las expresiones de su cuerpo.

Llamadas a función: Representadas mediante un nodo (de tipo *call*) enlazado con los grafos de todos sus argumentos y con otro nodo (de tipo *return*) que representa el valor de retorno de la llamada.

Cláusulas: Son usadas tanto en las funciones como en las instrucciones `if` y `case`: Se representan con un nodo (de tipo *clause_in*) enlazado a los grafos de sus patrones y a otro nodo (de tipo *guards*) que representa las guardas, y que al mismo tiempo está enlazado con el grafo del cuerpo de la claúsula (cuya raíz es un nodo de tipo *body*). La diferencia fundamental entre cláusulas de función, de `if` o de `case`, es que la instrucción `if` no tiene patrones, la instrucción `case` tiene un patrón, y la función tiene tantos patrones como parámetros tenga.

Instrucciones if y case: Se representan con un nodo (de tipo *if* o *case*) enlazado con los grafos de las distintas cláusulas. En el caso de la instrucción `case`, el nodo se enlaza además con el grafo de la expresión a evaluar.

Definiciones de función: Representadas con un nodo raíz de la función (nodo de tipo *function_in*) enlazado con los grafos de todas sus cláusulas.

Ejemplo 1 *El EDG de la Figura 1(a) representa el siguiente código Erlang:*

```
X={1,2},
{A,B}=X.
```

2.2. Representación y tipos de dependencias en el EDG

Las dependecias entre los componentes de un programa se representan en el EDG mediante arcos. Existen varios tipos de arcos, cada uno representa un tipo distinto de dependencia:

Arcos de Control: Denotan la dependencia producida en el flujo de la ejecución del código. Este tipo de arco es el utilizado en la representación de las estructuras sintácticas descrito en la Sección 2.1 para unir los distintos componentes.

Arcos de Datos: Muestran las relaciones de flujo de datos. Se observan cuatro casos:

i) Dependencia declaración-uso: Se une el nodo donde toma valor una variable con cada uno de sus usos.

ii) Dependencias producidas por ajuste de patrones: En el ajuste de patrones, estos arcos se usan para emparejar estructuralmente los nodos de la expresión con los respectivos nodos de los componentes del patrón.

iii) Dependencias producidas por restricciones impuestas por patrones: En las cláusulas, se introduce un arco entre el nodo de un patrón (asociado a un parámetro) y el nodo *clause_in* cuando este patrón impone una restricción implícita en dicha cláusula (patrones con variables repetidas y patrones con literales).

iv) Dependencias producidas por llamadas a funciones: En las llamadas a funciones se unen los nodos que representan las funciones a aplicar con el nodo *call* de la llamada.

Arcos de Entrada/Salida: Representan flujo de información entre distintas funciones. Estos arcos conectan la llamada y las posibles cláusulas correspondientes con las que se puede enlazar (dependiendo del ajuste de patrones entre los parámetros de la llamada y los argumentos de la cláusula).

- *Los arcos de entrada* unen cada argumento de la llamada con su parámetro correspondiente en la declaración. Para tratar el caso en que la función no tenga parámetros se añade otro arco de entrada entre el nodo de la llamada (*call*) con el nodo de la cláusula (*clause_in*).

- *Los arcos de salida* enlazan los nodos que definen el valor de retorno (última expresión del cuerpo de la cláusula) con el nodo *return* de la llamada.

Arcos de Resumen: Se utilizan para capturar dependencias entre funciones de forma precisa. Éstas indican qué argumentos influencian en el resultado devuelto en una llamada a función. Para ello, enlazan los nodos de estos argumentos con el nodo *return* de la llamada.

2.3. Algoritmo de fragmentación basado en el EDG

Una vez construido el EDG, obtener fragmentos a partir del mismo tiene un coste lineal con respecto a su tamaño [Sil12]. El algoritmo es igual al usado en lenguajes imperativos [HRB90], recibe como entrada un nodo seleccionado por el criterio de fragmentación y consta de dos fases:

1. Selecciona todos los nodos alcanzables (en cualquier orden) desde el criterio de fragmentación recorriendo hacia atrás arcos de control, datos, resumen y entrada.
2. Selecciona todos los nodos alcanzables (en cualquier orden) desde los nodos seleccionados en la primera fase siguiendo hacia atrás arcos de control, datos, resumen y salida.

2.4. Limitaciones del EDG estándar

Una limitación del EDG estándar es la pérdida de precisión producida en el tratamiento del ajuste de patrones. Las listas y tuplas se usan en Erlang para almacenar colecciones de elementos. De esta forma, este tipo de valores compuestos se pueden guardar en una variable y, posteriormente, ser extraídos del mismo. Como puede verse en el siguiente ejemplo, el problema de la

pérdida de precisión se produce cuando, al aplicar el algoritmo y atravesar estas variables, se incluyen en el fragmento todos los valores guardados en dichas variables, tanto los deseados como los no deseados.

Ejemplo 2 *Considérese el código del Ejemplo 1 y su EDG asociado de la Figura 1(a). Si escogemos como criterio de fragmentación el nodo que representa la variable* `A` *y aplicamos el algoritmo de fragmentación obtenemos como fragmento el conjunto de nodos oscuros. Obsérvese que en el fragmento se incluye la variable* `X` *de la primera instrucción. Esto implica la inclusión en el fragmento de los nodos que representan los valores 1 y 2. Sin embargo, nótese que el valor asociado a la variable* `A` *es únicamente el 1, y no el valor 2 de la tupla que también ha sido incluido, produciendo así una pérdida de precisión.*

Desde un punto de vista técnico, el problema se produce por la transitividad en la dependencia de datos. El EDG indica que el valor de `A` depende del valor de `X`, y éste a su vez depende de 1 y 2. Una representación más precisa debería indicar que el valor de `A` depende de *una parte de* `X`.

3. Aumento de precisión en el EDG

La pérdida de precisión observada en el Ejemplo 2 debe ser tenida en cuenta puesto que en Erlang, al igual que en otros lenguajes declarativos, es muy común el plegado y desplegado de las estructuras de datos formadas por tuplas o listas mediante el uso del ajuste de patrones. Por este motivo, en esta sección, exponemos una nueva definición del EDG en la que se da solución a esta pérdida de precisión. Primero se muestra cómo hemos mejorado el EDG en el caso intraprocedural mediante la incorporación de un nuevo tipo de arco llamado *Arco de Dato Parcial*. A continuación, el caso interprocedural se trata de forma similar con la incorporación de dos nuevos tipos de arcos llamados *Arco de Dato Parcial de Entrada* y *Arco de Dato Parcial de Salida*. Cabe destacar que todas estas modificaciones son conservativas respecto a la versión anterior, permitiendo generalmente aumentar la precisión de los fragmentos.

3.1. Mejora del EDG intraprocedural

Para solucionar la pérdida de precisión observada en el Ejemplo 2, definimos un nuevo tipo de arco al que denominamos *Arco de Dato Parcial* que representa una dependencia parcial de datos. Concretamente, este nuevo tipo de arco indica que se está desplegando una variable que contiene una estructura compuesta (una lista o una tupla). Así, este arco une los nodos de los patrones donde se hace el desplegado con los nodos asociados a las variables que contienen la estructura plegada. Además, para que el nuevo algoritmo de fragmentación funcione correctamente, se introduce un nuevo caso en el que se añaden arcos de datos. De esta forma, se unen mediante arcos de datos los nodos que representan la estructura en la que se produce un desplegado con sus correspondientes valores originales. Obsérvese por tanto que dado un nodo del EDG que pertenece a una estructura de datos compuesta, (1) utilizaremos un arco de datos para enlazar éste con los nodos de los que se obtienen los valores de ese nodo, y (2) utilizaremos arcos de dato parcial para enlazar el nodo con todos aquellos otros nodos que contienen la estructura de datos.

Por último, se ha añadido una regla al algoritmo de fragmentación para evitar incluir en el fragmento los nodos en los que no estamos interesados. Para ello, cuando se ha atravesado (en sentido inverso) un Arco de Dato Parcial, el nodo alcanzado se añade al fragmento pero ya no se continúa atravesando arcos a partir de éste.

La Figura 1(b) muestra el EDG del código del Ejemplo 1 con los Arcos de Dato Parcial con una linea fina continua azul. Podemos observar que las dos apariciones de la variable `X` se han enlazado mediante Arcos de Dato Parcial con los distintos elementos de la estructura `{A,B}` donde se está desplegando su valor. Además, los nodos de la estructura `{A,B}` se han enlazado mediante arcos de datos con los nodos de los valores correspondientes en la estructura `{1,2}`. Obsérvese en la figura el arco de dato parcial que une los nodos que representan la variable `X` de la primera expresión y la variable `A`. Tras recorrer este arco ya no se sigue recorriendo ningún arco desde el nodo que representa la variable `X` y, por tanto, se evita incluir el nodo que representa el valor 2. Nótese también que gracias a los nuevos arcos de datos el nodo que representa la variable `A` está enlazado directamente con el nodo que contiene su valor (el término 1 de la primera expresión). Comparando los EDG de las Figuras 1(a) y 1(b) podemos observar el aumento de precisión que esto representa.

Los Arcos de Dato Parcial junto con los nuevos arcos de datos que unen las distintas expresiones implicadas en un ajuste de patrones, consiguen capturar única y exclusivamente los valores de la estructura en los que estamos interesados.

3.2. Mejora del EDG interprocedural

En la Sección 3.1 hemos visto cómo se trata el plegado y desplegado de valores dentro de una misma función, pero ¿qué ocurre si dicho plegado/desplegado se realiza en el paso de parámetros o en los valores de retorno de funciones diferentes? Teniendo en cuenta que este recurso es muy utilizado por la gran flexibilidad y oportunidades que brindan al programador, es importante que se trate de forma eficiente. Así, el caso interprocedural se trata de forma similar al caso intraprocedural pero diferenciando entre el caso en el que el desplegado se produce en los parámetros de la función y en el que se produce en el valor de retorno de la función.

Para explicar los nuevos arcos introducidos en esta sección utilizaremos los EDG de las Figuras 2 y 3 que se han obtenido respectivamente a partir de los Ejemplos 3 y 4. En estos ejemplos el criterio de fragmentación aparece encuadrado.

Ejemplo 3

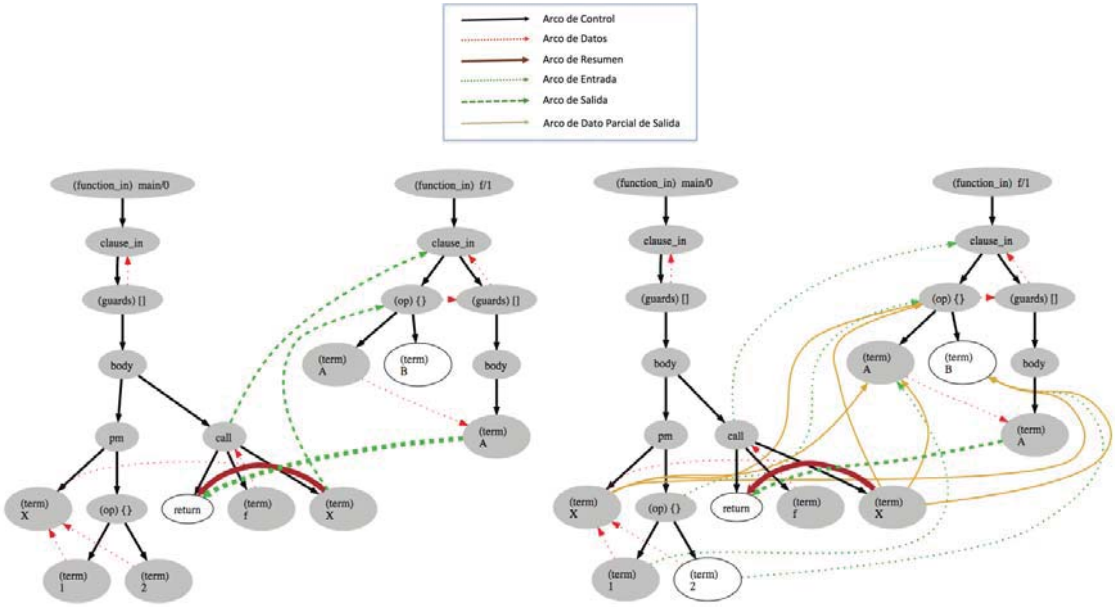
```
main() -> X = {1,2},
          f(X).
f({A,B}) -> [A].
```

Ejemplo 4

```
main() -> {[A],B}=f().
f() -> X={1,2},
       X.
```

3.2.1. Arcos de Dato Parcial de Entrada

Para tratar el caso en el que el desplegado se produce en los argumentos de entrada de una función, hemos añadido los *Arcos de Dato Parcial de Entrada* basándonos en la misma idea que en los Arcos de Dato Parcial. Así, estos arcos enlazarán siempre nodos de funciones distintas (o de contexto de llamada distinto si se trata de una llamada recursiva). En este caso también se enlazan los elementos de la estructura desplegada con su correspondiente valor, pero ahora mediante arcos de entrada. Con respecto al algoritmo, los Arcos de Dato Parcial de Entrada se



(a) EDG (b) EDG con los Arcos de Dato Parcial de Entrada

Figura 2: EDG asociados al código del Ejemplo 3.

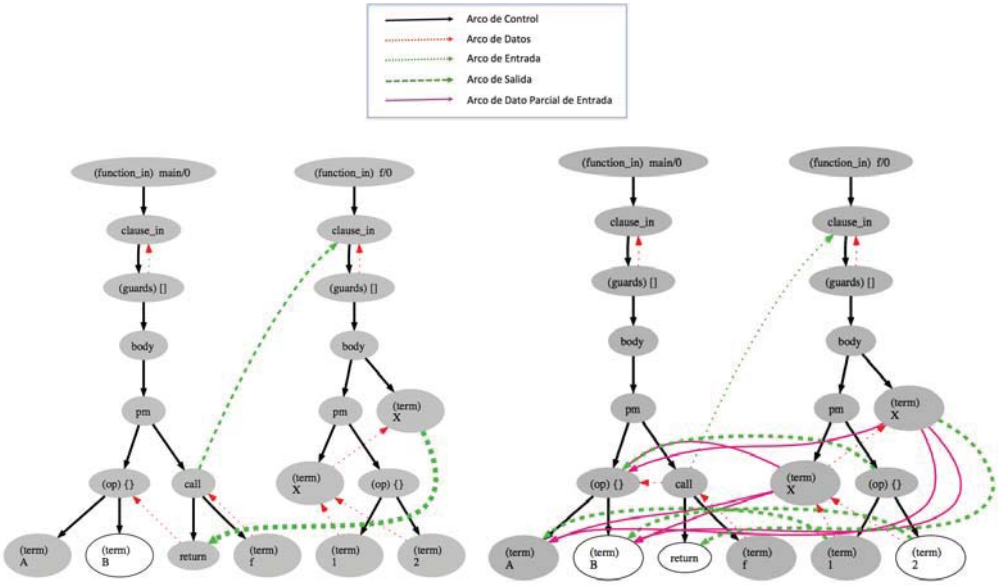
tratan de forma análoga a la descrita para los Arcos de Dato Parcial, excepto que sólo se recorren en la primera fase del algoritmo (cuando se atraviesan los arcos de entrada).

En los EDG de la Figura 2, asociados al código del Ejemplo 3, se muestran los fragmentos obtenidos (nodos oscuros) tanto en la versión original del EDG (Figura 2(a)) como en la nueva con las modificaciones especificadas (Figura 2(b)). Puede observarse como las modificaciones introducidas han provocado que el nodo que representa el valor `2` no se incluya en el fragmento resultante.

3.2.2. Arcos de Dato Parcial de Salida

Para el caso en el que el desplegado se realiza sobre el valor de retorno de una función hemos introducido los *Arcos de Dato Parcial de Salida*. Por tanto estos arcos son análogos a los Arcos de Dato Parcial. Además, el algoritmo es el mismo teniendo en cuenta únicamente que los Arcos de Dato Parcial de Salida sólo se recorren en la segunda fase del algoritmo (en la que se atraviesan los arcos de salida).

En los EDG de la Figura 3, asociados al código del Ejemplo 4, puede verse que la introducción de estos nuevos arcos (Figura 3(b)) ha provocado que el nodo que representa el valor `2` no se haya incluido en el fragmento resultante. En la Figura 3(a) se observa cómo en la versión original del EDG este nodo sí se ha incluido en el fragmento.



(a) EDG (b) EDG con los Arcos de Dato Parcial de Salida

Figura 3: EDG asociados al código del Ejemplo 4.

4. Conclusiones

En algunas aplicaciones (e.g., compiladores, depuradores, etc.) es esencial obtener una buena precisión en los fragmentos obtenidos al aplicar la técnica de fragmentación. En el marco de los lenguajes funcionales, el plegado y desplegado de valores compuestos es una técnica muy utilizada. Por tanto, aumentar la precisión de los fragmentos en estos casos puede suponer una mejora considerable. Precisamente, ésta ha sido la principal aportación de este trabajo en el que, a partir de la definición previa del EDG [STT12] y mediante la introducción de nuevos tipos de arcos y pequeñas modificaciones en el algoritmo, se ha podido mejorar esta precisión tanto en el caso intraprocedural como en el caso interprocedural. Se ha implementado una herramienta (*Slicerl*) para aplicar la técnica de fragmentación a programas Erlang donde también pueden verse los EDG generados. La herramienta es pública y puede encontrarse en [STTI11].

Referencias

[FOW87] J. Ferrante, K. Ottenstein, J. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems* 9(3):319–349, 1987.

[FRT95] J. Field, G. Ramalingam, F. Tip. Parametric Program Slicing. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL '95, pp. 379–392. ACM, New York, NY, USA, 1995.

[HRB90] S. Horwitz, T. Reps, D. Binkley. Interprocedural Slicing Using Dependence Graphs. *ACM Transactions Programming Languages and Systems* 12(1):26–60, 1990.

[OSV04] C. Ochoa, J. Silva, G. Vidal. Dynamic Slicing Based on Redex Trails. In *Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*. PEPM '04, pp. 123–134. ACM, New York, NY, USA, 2004.

[RT96] T. Reps, T. Turnidge. Program Specialization Via Program Slicing. In *Proceedings of the Dagstuhl Seminar on Partial Evaluation, volume 1110 of Lecture Notes in Computer Science*. Pp. 409–429. Springer-Verlag, 1996.

[Sil12] J. Silva. A Vocabulary of Program Slicing-Based Techniques. *ACM Computing Surveys* 44(3), 2012.

[STT12] J. Silva, S. Tamarit, C. Tomás. System Dependence Graphs in Sequential Erlang. In *Fundamental Approaches to Software Engineering - 15th International Conference, FASE 2012*. Lecture Notes in Computer Science 7212, pp. 486–500. Springer, 2012.

[STTI11] J. Silva, S. Tamarit, C. Tomás, D. Insa. Slicerl. September 2011. http://kaz.dsic.upv.es/slicerl

[Tip95] F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages* 3(3):121–189, 1995.

[Wei81] M. Weiser. Program Slicing. In *Proceedings of the 5th international conference on Software engineering*. ICSE '81, pp. 439–449. IEEE Press, Piscataway, NJ, USA, 1981.

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): Actas de las "XII Jornadas sobre Programación y Lenguajes (PROLE'2012), Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

Sistema funcional distribuido de publicidad para iDTV

Macías López[1], **Laura M. Castro**[2], **David Cabrero**[3*]

[1] macias.lopez@madsgroup.org
[2] laura.castro@madsgroup.org
[3] david.cabrero@madsgroup.org
Grupo MADS – Departamento de Computación
Universidade da Coruña (A Coruña, España)

Abstract: Al diseñar un sistema distribuido, buenas prácticas como el uso de arquitecturas modulares o la aplicación de patrones de diseño son siempre deseables, pero hay otros aspectos también relevantes que pueden pasar inicialmente desapercibidos. Entre ellos, hay una serie de decisiones que deben tomarse acerca de las comunicaciones entre los nodos del sistema: el formato de los mensajes que se han de enviar, las características deseadas de la red (latencia, ancho de banda . . .), etc.

Uno de los elementos más críticos en el diseño e implementación de sistemas distribuidos es la definición de un buen tratamiento de la caída de nodos y *netsplits*. La situación puede llegar a ser realmente delicada si este tipo de contingencias se detectan una vez que el sistema está en producción, por lo que es muy importante definir los mecanismos apropiados para tratarlas: elegir cuáles y cómo ponerlos en práctica no sólo depende de la tecnología utilizada, sino también de la fiabilidad de la red de comunicaciones, o incluso del hardware en el que se ejecuta el sistema.

En este trabajo presentamos ADVERTISE, un sistema funcional distribuido para la transmisión de publicidad a los set-top-box (STB) de los hogares a través de una red de televisión digital (IDTV) de una operadora de cable. Hemos usado este sistema como un caso de estudio para explicar la forma en que se abordaron los problemas antes mencionados.

Keywords: sistemas distribuidos, tolerancia a fallos, caída de nodos, *netsplit*, consistencia, disponibilidad.

1. Introducción

El diseño, e implementación de sistemas distribuidos es una tarea compleja [CPV97, KDK+89], que involucra mucho más que la aplicación de un determinado modelo de concurrencia. Para asegurar el correcto funcionamiento de este tipo de sistemas, hemos de meditar cuidadosamente las decisiones que suelen presentarse en estos escenarios, así como los problemas que más frecuentemente suelen acarrear.

Uno de los problemas más comunes en el desarrollo de sistemas distribuidos es la definición de una política apropiada para la gestión de caídas de nodos y particiones de la red (*netsplits*) [BBMS08, SJ05], un problema que, en muchos casos, sólo aparece una vez que el sistema

* Trabajo parcialmente financiado por el proyecto EU FP7 OptiBand (grant n. 248495), y MICIN TIN2010-20959.

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): PROLE'2012, pp. 277-291, ISBN:978-84-15487-27-2.
Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

se encuentra ya en fase de despliegue. Existen mecanismos de contingencia significativamente diferentes que se pueden proponer para afrontar este tipo de situaciones [SHB04]: elegir cuál(es) y cómo aplicarlo(s) depende no sólo de la tecnología utilizada, sino también de la fiabilidad de la red de comunicaciones, o incluso del hardware en el que se ejecuta el sistema. Además, cuando se trata con estos problemas, también es importante analizar las garantías que las distintas soluciones proporcionen en cuanto a la consistencia de datos y/o su disponibilidad [Bre00]. Una buena solución será, en cada caso, aquélla que nos permita asegurar la integridad de los datos manejados por el sistema a la vez que minimizar las consecuencias inesperadas e indeseadas de los fallos en la comunicación entre nodos.

Para estudiar este problema, hemos utilizado un caso de estudio industrial: ADVERTISE, un sistema distribuido para la transmisión de publicidad a los set-top-boxes (STB) en el hogar a través de una red de televisión digital (IDTV) de una operadora de cable. El sistema, construido en Erlang [CT09, Arm07], debe garantizar la adecuada coordinación de los mecanismos de publicidad: la compilación de eventos, la emisión de señales de publicidad para los STB durante un período de tiempo, el registro del número de veces que un elemento publicitario específico se muestra, etc. El desafío principal en este caso de estudio fue la gestión del tamaño de la red de comunicaciones, en concreto su alto – y creciente – número de usuarios (esto es, los hogares, los clientes de la operadora, que rondan los 100.000 en el momento de escribir este trabajo). Para soportar esta carga, ADVERTISE se diseña como un sistema compuesto por varios nodos, y para lograr los objetivos, resultó necesario implementar mecanismos de tolerancia a fallos para hacer frente a los problemas antes mencionados, abordándolos desde el punto de vista de la programación funcional.

En este trabajo describimos el funcionamiento general y la arquitectura de ADVERTISE, junto con la aplicación de los mecanismos Erlang que fueron elegidos para garantizar la integridad del sistema y sus datos, reduciendo al mínimo los efectos no deseados producidos por los problemas de intercomunicación de nodos. ADVERTISE es un sistema actualmente en producción, en operación ininterrumpida, que se ejecuta en hardware con características inferiores a lo esperado, lo que en nuestra opinión, habría sido imposible sin la aplicación de las soluciones mencionadas.

El artículo está estructurado de la siguiente manera: en la Sección 2, describimos los desafíos y problemas habituales en el diseño de sistemas distribuidos, así como algunas consideraciones al respecto teniendo en cuenta el uso de Erlang como plataforma de desarrollo y despliegue. La Sección 3 contiene información detallada sobre ADVERTISE, sus componentes, funcionalidades y tareas. En la Sección 4, exponemos la arquitectura distribuida basada en Erlang que diseñamos para ADVERTISE, y explicamos los mecanismos de contingencia implementados. Terminamos presentando nuestras conclusiones en la Sección 5.

2. Arquitecturas software distribuidas

A medida que el software se vuelve más y más complejo, y aborda tareas y necesidades cada vez más exigentes, nos vemos obligados a resolver problemas que van más allá de la funcionalidad de una aplicación dada, y tienen que ver con requisitos no funcionales y criterios de usabilidad como los tiempos de ejecución y respuesta, la seguridad, la fiabilidad, la robustez, o la tolerancia a fallos. Una de las maneras en que analistas y desarrolladores pueden abordar este nuevo

abanico de requisitos consiste en la migración de dichas aplicaciones a un ecosistema distribuido multiproceso. Sin embargo, las grandes capacidades de las arquitecturas distribuidas traen como contrapartida su propio conjunto de problemáticas que debemos afrontar [WJ99, NT93].

Una de las principales cuestiones a tener en cuenta al diseñar e implementar una arquitectura de software distribuida es la accesibilidad de los nodos [TLLS09]. Cuando un nodo en un sistema distribuido no puede ser alcanzado en un momento dado, es muy difícil conocer la causa específica para este comportamiento: un fallo de hardware, un error de la aplicación, congestión de la red, *netsplit*, etc. En algunos de estos casos, la aplicación ya no se ejecuta en el nodo inalcanzable, pero si hay otros nodos en funcionamiento, sí es probable que la aplicación se siga ejecutando en ellos. Sin embargo, en otros casos, el nodo inalcanzable aún puede estar ejecutando la aplicación, y en ese caso, son todos los demás nodos los que están caídos, bajo su punto de vista. El proceso de reincorporación del nodo al clúster difiere radicalmente en uno y otro caso, y ambos escenarios han de ser modelados apropiadamente.

2.1. El teorema CAP

En el año 2000, el profesor Eric Brewer presentó la conjetura CAP [Bre00] (demostrada dos años más tarde [GL02]), que establecía que un sistema distribuido en el que se compartían datos sólo podía presentar al mismo tiempo dos de estas tres propiedades:

Consistencia (*Consistency*) La consistencia en un sistema distribuido en el que se comparten datos implica que todas las operaciones sobre dichos datos deben completarse *atómicamente*. Esta *consistencia linearizable* significa que no puede haber dos operaciones diferentes utilizando los mismos datos y modificándolos al mismo tiempo. De lo contrario, diferentes partes del sistema (i.e. nodos) podrían producir versiones diferentes, y por tanto inconsistentes, de los mismos datos, siendo imposible obtener una ordenación secuencial de dichas operaciones que arrojase el mismo resultado [GL02]. Típicamente, la consistencia se consigue aplicando mecanismos de transaccionalidad, donde todos los nodos deben ponerse de acuerdo a la hora de consolidar las modificaciones sobre los datos.

Disponibilidad (*Availability*) Cuando hablamos de la disponibilidad de un sistema, nos referimos al hecho de que, siempre que un cliente envíe una petición, el sistema debe ser capaz de devolver una respuesta. Originalmente, no se consideraba relevante el tipo de respuesta, sino sólo el hecho se devolviese alguna. No obstante, hoy muchos expertos consideran que dicha respuesta debería contener información útil para considerar que el sistema tiene esta propiedad [GL02]. Esto significa que si el sistema no da respuesta a una solicitud, o bien devuelve una respuesta del tipo '*timeout*' o '*sistema no disponible, vuelve a intentarlo más tarde*', no satisface el de disponibilidad.

Tolerancia a particiones (*Partition-tolerance*) Esta propiedad es la piedra angular del teorema CAP, y representa la capacidad de funcionamiento o recuperación que un sistema presenta ante particiones de la red. Ante un fallo de la red (ya sea total o parcial), es decir, cuando algunos de sus componentes (i.e. nodos) no puedan comunicarse (o lo hagan con mucha dificultad –latencia), un sistema tolerante a particiones podría seguir funcionando. Si un

sistema distribuido no implementa mecanismos para lidiar con este tipo de situaciones, significará que si la red pierde mensajes o los nodos no pueden enviarlos o recibirlos, entonces no habrá ninguna garantía sobre cuál de las dos propiedades anteriores se conservará.

En la práctica, sólo hay dos combinaciones posibles de las anteriores propiedades que son factibles de acuerdo con el teorema CAP: consistencia + tolerancia a particiones o disponibilidad + tolerancia a particiones. No hay ninguna tecnología ni topología que nos asegure que no van a producirse fallos de comunicación, por lo que la tolerancia a particiones siempre debe considerarse como parte del diseño de un sistema distribuido. Y durante una partición de red, un sistema puede permanecer disponible o consistente, pero no ambas a la vez.

Así, a la hora diseñar e implementar un sistema, deberemos elegir entre hacer prevalecer la coherencia de los datos compartidos pagando el precio de que algunos nodos no estén disponibles durante las particiones de la red, o bien hacer prevalecer la disponibilidad de los nodos pagando el precio de operar con datos potencialmente inconsistentes. Un enfoque que prime disponibilidad sobre consistencia será generalmente más flexible, pero ambas opciones son igualmente legítimas, siempre y cuando se alineen con las necesidades y especificaciones del sistema.

2.2. El lenguaje funcional distribuido Erlang

Erlang [LMC10] es un lenguaje de programación funcional diseñado con especial atención a la concurrencia y la distribución. En terminología Erlang, un sistema distribuido está compuesto por una serie de nodos que se comunican a través de una red. Cada nodo puede ejecutar múltiples procesos ligeros que tienen su propio espacio de memoria (que no comparten).

Los procesos Erlang usan paso de mensajes asíncrono, que se implementa con un buzón de mensajes por proceso. El buzón de cada proceso está siempre listo para recibir mensajes, y los procesos pueden usar la potente sintaxis declarativa, basada en *pattern-matching*, para leer mensajes del buzón en un orden diferente al de llegada.

Dos o más procesos Erlang se pueden *enlazar* de manera que, si uno falla, el resto de procesos enlazados recibirá un mensaje de notificación. Gracias a este mecanismo, en Erlang pueden crearse jerarquías de supervisión de procesos, donde algunos se dedican a hacer el trabajo 'real' y otros se ocupan de supervisar a los anteriores, detenerlos y reiniciarlos en caso de ser necesario.

Los diseñadores de Erlang tomaron la decisión de considerar los nodos inalcanzables como caídos, y por lo tanto sólo los nodos que pueden ser contactados se consideran activos. Esta perspectiva tiene sentido cuando necesitamos respuestas rápidas ante caídas del sistema, ya que asume que la red tiene menos probabilidades de fallar que el software o el propio hardware. Ésta, de hecho, era la realidad de la inmensa mayoría de los sistemas en los que Erlang se utilizó originalmente. Haber adoptado otro punto de vista, en el que los nodos inalcanzables se supusieran aún activos y se tratase de hacer frente a tal contingencia, hubiese derivado en mecanismos de tolerancia a fallos más lentos, obligando al sistema a esperar y hacer frente a la posible reintegración de los nodos presuntamente caídos tras la recuperación de la conectividad.

Sin embargo, hoy en día la mayoría de sistemas distribuidos, y entre ellos los implementados utilizando Erlang, han cambiado significativamente de naturaleza y hacen en su mayoría un uso intensivo de redes de comunicaciones relativamente poco fiables, y por lo tanto, se enfrentan a un importante número de fallos debidos a problemas de inaccesibilidad. En los sistemas distri-

buidos Erlang, además, nos enfrentamos al problema de la reintegración de los nodos que en algún momento se han considerado caídos, pero de repente vuelven a estar disponibles porque en realidad nunca presentaron un fallo software o hardware, sino que eran simplemente inalcanzables durante un tiempo limitado debido a interrupciones en la red, congestión, etc. No ser capaces de hacer frente adecuadamente a este tipo de contingencias puede dar lugar a situaciones muy indeseables en el funcionamiento de un sistema distribuido: que un nodo vuelva a conectar con un clúster tras haber estado operando normalmente mientras permanecía aislado del grupo, puede plantear un problema de consistencia de datos importante, ya que todo el sistema puede haber estado manejando datos obsoletos.

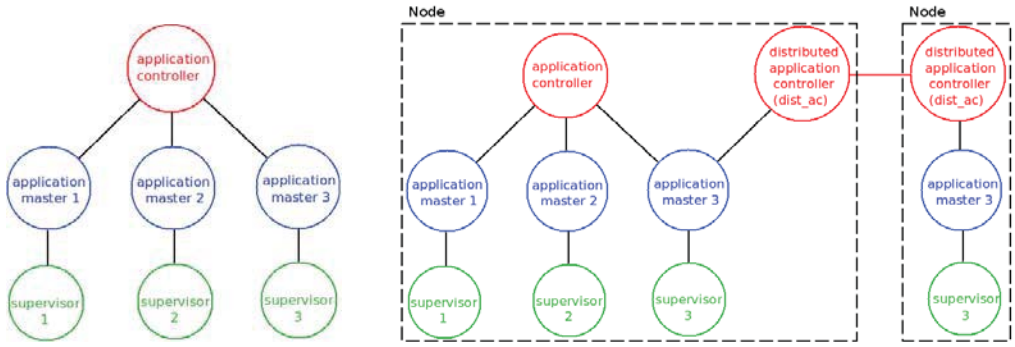
2.2.1. Arquitecturas distribuidas en Erlang

En Erlang, un nodo es una instancia de una máquina virtual Erlang (EVM). Los nodos Erlang pueden ejecutarse en diferentes máquinas, o incluso coexistir varios en la misma máquina física. Cuando se arranca un nodo, se le asigna un nombre y se conecta a una aplicación llamada EPMD (Erlang Port Mapper Daemon), que se ejecuta en cada máquina física que forma parte de un clúster Erlang. La EPMD se inicia cuando arranca la primera EVM, y actúa como un servidor de nombres que gestiona posibles conflictos.

Un nodo puede establecer una conexión a otro nodo. Cuando esto sucede, los dos nodos inician una monitorización mutua, de manera que ambos son conscientes de cuando se interrumpe la comunicación entre ellos. Cada vez que un nuevo nodo se une al grupo, todos ellos se interconectan de esta manera. Por supuesto, a pesar de estar todos conectados, cada nodo actúa de manera forma independiente: administra su propio conjunto de los procesos registrados, sus propias ETS (tablas de almacenamiento), módulos cargados, etc. Los nodos que están conectados pueden intercambiar mensajes; más concretamente, los procesos que se ejecutan en nodos que están conectados pueden enviarse mensajes. No hay diferencia alguna entre el envío de un mensaje a un proceso local (es decir, un proceso en el mismo nodo) o a un proceso remoto (en otro nodo, en la misma máquina física o en otra). Los mensajes se intercambian de manera transparente, incluyendo la *serialización* de las estructuras de datos que puedan incluir, que se mantendrán válidas tanto en el nodo local como en uno remoto. Esto permite, por ejemplo, enviar identificadores de procesos a través de la red y utilizarlos para establecer nuevos vínculos.

Cuando se arranca una EVM se arranca también un proceso llamado Controlador de Aplicaciones (CA). El CA actúa como un supervisor de todas las aplicaciones Erlang que se ejecutan en ese nodo, siempre y cuando cumplan con el *behaviour* `application`. Un *behaviour* en Erlang es la aplicación de un principio de diseño, un patrón común que se ofrece como librería y permite obtener cierto comportamiento simplemente implementando un conjunto mínimo de funciones de *callback*. En particular, el *behaviour* `application` ofrece a un conjunto de componentes (módulos o procesos) la capacidad de arrancarse y detenerse en bloque. Cuando se usa este *behaviour*, el CA arranca un proceso llamado Maestro de Aplicación (MA), que es una capa intermedia entre el CA y el supervisor principal de cada aplicación específica. La figura 1(a) representa la ejecución de tres aplicaciones en una misma EVM.

Las aplicaciones Erlang distribuidas, además de con el esquema de la figura 1(a), son controladas por un segundo CA denominado CA Distribuido (Fig. 1(b)). Esto permite que los nodos en los que una aplicación distribuida se va a ejecutar mantengan contacto entre sí y negocien



(a) EVM: Controlador de Aplicaciones

(b) Varias EVM: CA Distribuido

Figura 1: Aplicaciones Erlang

cuál de ellos la arrancará utilizando una lista de prioridades. Cada nodo arranca al iniciarse una instancia del CAD, y los CADs están en contacto unos con otros.

Una aplicación Erlang distribuida puede estar, en un nodo dado, en dos estados diferentes: `started` o `running`. La diferencia entre estos estados es que, aunque una aplicación se puede iniciar en todos los nodos de un clúster Erlang, sólo se puede ejecutar realmente en uno de ellos. En otras palabras, lo que se distribuye entre los múltiples nodos Erlang es la *gestión* de la aplicación. Los nodos que no están ejecutando la aplicación permanecen suspedidos, y entran en acción en caso de que el nodo donde se ejecuta la aplicación falle. En ese momento, uno de ellos toma el relevo y continúa ejecutando la aplicación.

Para entender los mecanismos de tolerancia a fallos, introducimos los conceptos de *failover* y *takeover*. Un *failover* es la situación que acabamos de describir, en la que una aplicación se reanuda en un nodo diferente cuando aquél en el que se estaba ejecutando falla por cualquier razón. Un *takeover*, por otro lado, se produce cuando se recupera un nodo que falló mientras ejecutaba una aplicación y, en lugar de dejar que el que lo reemplazó continúe en operación, exige la 'devolución' de la ejecución. Los *takeover* tienen sentido si existe algún criterio de prioridad entre los nodos. Si la configuración indica que un nodo recuperado (o incluso uno nuevo que acaba de incorporarse al clúster) tiene una prioridad más alta (porque, por ejemplo, se está ejecutando sobre mejor hardware), entonces la ejecución de la aplicación se transferirá del nodo que la está ejecutando en ese momento al nodo con mayor prioridad.

En nuestro caso de estudio hemos implementado dos mecanismos de seguridad: uno que comprueba la integridad de los nodos periódicamente (para comprobar que los nodos activos siguen siendo los indicados en la configuración), y otro que gestiona los *failovers* y *takeovers* cuando se detecta una partición en la red. Ambos se explican en detalle en la sección 4.

3. Caso de estudio: ADVERTISE

En el contexto de los operadores de televisión por cable, el catálogo de productos que ofrecen a sus clientes incluye habitualmente una gama de servicios que proporcionan un escenario ade-

cuado para la transmisión de publicidad. El uso óptimo de la conexión por cable es, pues, una fuente rentable de ingresos para el operador, y un sistema para la transmisión de publicidad que no sólo inserte anuncios de manera oportuna y eficiente, sino que también tenga en cuenta las características y preferencias del cliente, representa sin duda un inteligente aprovechamiento.

Estos requisitos condujeron al desarrollo del proyecto ADVERTISE, un sistema para la transmisión de publicidad a los hogares a través de los *set-top-box* (STB) de los clientes de una red de televisión digital (IDTV) de un operador de cable. Para poder proporcionar mecanismos de tolerancia a fallos y cortes de red, ADVERTISE se diseña como un sistema distribuido.

3.1. Elementos y definiciones

Describimos a continuación algunos conceptos clave para la comprensión del caso de estudio:

- **Medio:** Término que se refiere a cualquier activo que muestra el STB. El sistema es compatible con varios tipos de medios: textos, imágenes, y medios interactivos.
- **Regla:** Cada uno de los medios que el sistema puede mostrar es manejado por una regla. Las reglas establecen parámetros de visualización tales como rangos de fechas y horas específicos, número de accesos (es decir, de veces que se muestra el medio), frecuencia, etc. Las reglas se organizan en campañas, pero la regla puede modificar y particularizar los valores que la campaña general da a los distintos parámetros.
- **Campaña:** Unidad organizativa de medios y reglas, que les proporciona coherencia y significado. La campaña establece límites y los valores por defecto para los medios que incluye.
- **Acción:** ADVERTISE traduce cada regla de una campaña a un formato entendible por los STB, denominado acción. Las acciones son, por lo tanto, las reglas procesadas por ADVERTISE antes de enviarse a los STB.

3.2. Arquitectura general

La figura 2 muestra una vista general de ADVERTISE, incluyendo sus principales componentes. La arquitectura incluye tres actores principales:

- **Servidor ADVERMAN**. Fachada de administración de ADVERTISE. Implementada en Java, se comunica con el servidor ADVERTISE y proporciona acceso a todas operaciones relacionadas con la publicidad: carga de medios, creación/actualización/eliminación de reglas, creación/activación/desactivación de campañas, etc. Este subsistema también genera informes y gráficos que muestran las estadísticas de las diferentes campañas.
- **Servidor ADVERTISE**. El núcleo del sistema de publicidad. Implementado en Erlang, gestiona todas las campañas, transforma las reglas en acciones, y se ocupa de todos los detalles relacionados con la activación de las publicidades y su envío a los STB.
- **Set-top-boxes (STB)s**. Los receptores finales de los medios publicitarios.

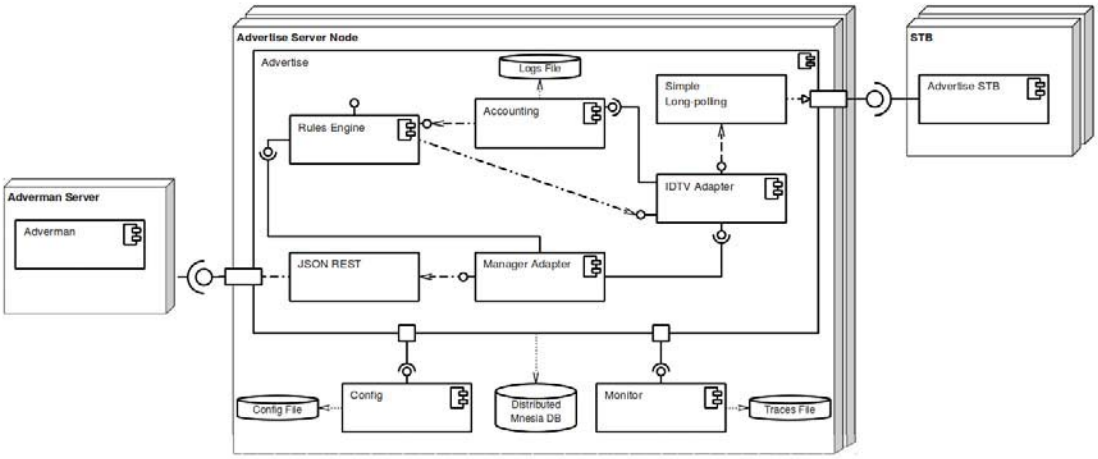


Figura 2: Arquitectura general de ADVERTISE

3.2.1. Módulos del servidor ADVERTISE

Los módulos principales del servidor ADVERTISE son:

- **Motor de Reglas**. Recibe y organiza campañas, reglas y medios. Entre todas las campañas y reglas determina, en cada momento, cuáles están activas (comprobando fechas de validez y otras restricciones), verifica que los medios están correctamente almacenados en el sistema, y construye las acciones que se enviarán a los STB, organizando su distribución y manteniendo registro de cuántas veces se muestra cada medio. La periodicidad con la que se comprueban y descartan reglas y campañas es configurable. Naturalmente, este módulo es único en el sistema y sólo se ejecuta en el nodo maestro.

- **Adaptador IDTV**. Actúa como intermediario entre el servidor ADVERTISE y los STB. Envía las acciones y medios a los STB, recibe de ellos información estadística que reenvía a los módulos correspondientes de ADVERTISE, y reparte la carga del sistema entre los diferentes nodos ADVERTISE físicos. Puesto que es el componente más cercano a los STB y tiene información de su estado (por ejemplo, el tipo de suscripción del cliente), realiza un filtrado de las acciones producidas por el motor de reglas en base a si son o no relevantes para los STB (es decir, para el perfil del cliente correspondiente), y evita el envío de medios no válidos o irrelevantes. Además, cuando se registra en el sistema un nuevo STB, le envía la lista de acciones actualmente en activo, por lo que necesita para almacenar una copia de dicha lista. La comunicación final con los STBs se realiza a través de un adaptador específico, cuya arquitectura permite manejar diferentes protocolos de petición/respuesta. En la actualidad, ADVERTISE se comunica con los STB intercambiando mensajes HTTP asíncronos a través de conexiones TCP permanentes establecidas sobre el canal de retorno (i.e. *long polling*), y la información se codifica utilizando JSON [Cro06] y XML.

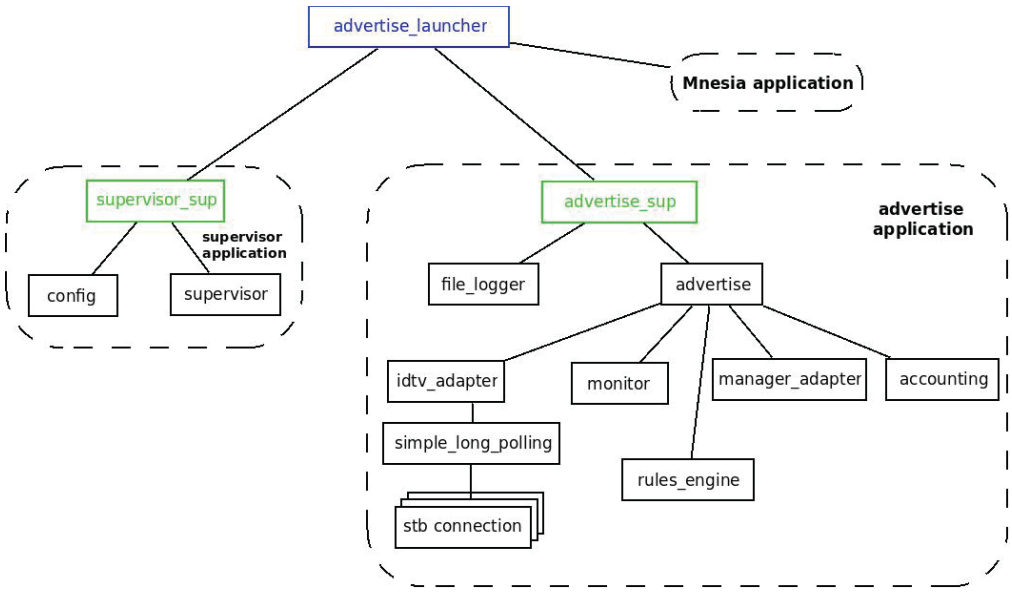


Figura 3: Árbol de supervisión de ADVERTISE

- **Manager Adapter**. Este módulo se comporta como un adaptador de interfaces: se encarga de transformar las distintas solicitudes que recibidas por el servidor ADVERMAN al servidor ADVERTISE (creación de campañas, actualización de reglas, carga de nuevos medios, etc.). Como ya se ha mencionado, la información sobre las campañas, reglas y medios se codifica utilizando JSON. Este adaptador supervisa el componente JSON REST [PZL08], un módulo que gestiona un conjunto de procesos que escuchan conexiones TCP en determinados puertos. Cuando cualquiera de esos procesos recibe una solicitud ADVERMAN, se encamina al adaptador para que llegue al motor de reglas una vez parseado y traducido su contenido a estructuras de datos entendibles por el motor de reglas.

- **Contabilidad (*Accounting*)**. Almacena todos los eventos relacionados con las acciones que el usuario del STB puede llevar a cabo: pulsar teclas, visualizar anuncios, etc. El resto de módulos pueden solicitarle dicha información (como ADVERMAN, por ejemplo, para su generación de estadísticas).

- **Configuración y monitorización**. Módulos de uso general para obtener/establecer información de configuración y guardar registro de las actividades del sistema, respectivamente.

3.2.2. Árbol de supervisión

La figura 3 muestra el árbol de supervisión diseñado para ADVERTISE. Se han modelado dos aplicaciones Erlang: `supervisor` y `advertise`. La última es el sistema de publicidad que acabamos de describir. La aplicación supervisora es una aplicación Erlang distribuida que gestiona la selección del nodo maestro, coordina el arranque de los nodos del clúster, y ejecuta los mecanismos de contingencia implementados.

4. Arquitectura distribuida basada en Erlang de ADVERTISE

Como hemos explicado antes, una aplicación Erlang distribuida arranca en todos los nodos del clúster, pero sólo se ejecuta en uno de ellos. Cuando ese nodo falla, el mecanismo de distribución de Erlang selecciona otro nodo del clúster para seguir ejecutando la aplicación en (*failover*).

Para cumplir con los requisitos de tolerancia a fallos del operador de cable (como se explicará en la sección 4.4), diseñamos nuestro caso de estudio ADVERTISE para ser desplegado sobre un *mínimo* de tres nodos. ADVERTISE podría funcionar en un único nodo, pero no sería muy tolerante a fallos; de igual modo, si permitiésemos que ADVERTISE se ejecutase en dos nodos, los mecanismos de contingencia diseñados serían poco efectivos.

Además, utilizamos Mnesia [MNW99], la base de datos distribuida de Erlang, para almacenar la información relevante de campañas, reglas y medios: así, cuando se ejecuta una operación de escritura sobre Mnesia, la modificación se propaga a todos los nodos del sistema. Para arrancar Mnesia correctamente en un clúster de nodos Erlang, todos ellos deben estar activos, así que necesitamos un mecanismo de arranque coordinado para inicializarlos apropiadamente de modo que aseguremos que todos los nodos tienen la última copia de los datos en el todo momento.

A continuación explicamos las diferencias entre el proceso de *inicialización* y el proceso de *arranque* de ADVERTISE.

4.1. Proceso de inicialización de ADVERTISE

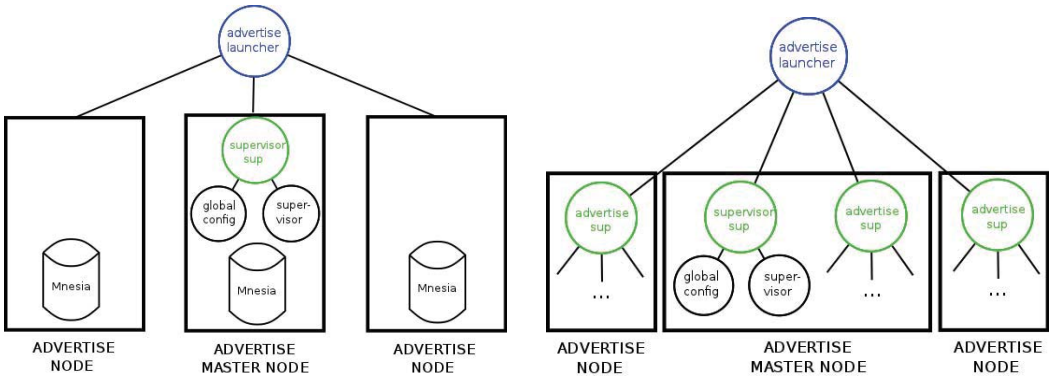
El proceso de inicialización de ADVERTISE se lanza en el nodo maestro, que lee de la configuración la lista de nodos que formarán parte del clúster y espera, monitorizándolos cada dos segundos, a que todos ellos estén arrancados. Una vez esto ocurre, inicializa el contenido de la base de datos Mnesia y se asegura de que el resto de nodos se sincronizan con dicho contenido. En particular, en el caso de ADVERTISE no hemos otorgado prioridades a los nodos, de manera que en caso de fallar el nodo maestro, cualquier otro nodo podría tomar el control de la aplicación de supervisión distribuida (CAD). Tras el proceso de inicialización, los nodos ADVERTISE están en la situación mostrada en la figura 4(a).

4.2. Proceso de arranque de ADVERTISE

El proceso de arranque de ADVERTISE, que puede lanzarse en cualquier nodo del cluster, comienza con la comprobación de si ADVERTISE está ya ejecutándose en algún otro nodo (situación que puede darse en caso de que el nodo se esté recuperando de un fallo). Si ADVERTISE ya se está ejecutando, el nodo enlaza su supervisor con la aplicación ADVERTISE, mecanismo que le permitirá realizar un *failover* en caso de que el nodo actualmente ejecutando la aplicación falle. Seguidamente se inicializan los mecanismos de contingencia, que consisten en:

1. **Comprobación de la integridad de los nodos** cada 10 segundos (sección 4.3.1).
2. **Comprobación de la aplicación distribuida** cada segundo (sección 4.4.1).

También se comprueba la integridad de Mnesia con respecto al nodo maestro, y la conectividad propia. Como último paso, en el caso en que se hubiese detectado que ADVERTISE no



(a) Tras el proceso de inicialización (b) Tras el proceso de arranque

Figura 4: Nodos ADVERTISE

se estaba ejecutando en ningún otro nodo, se arranca la aplicación ADVERTISE y todos los componentes que la integran (supervisor y servicios locales –`accounting`, `rule_engine`, `idtv_adapter` y `manager_adapter`). La figura 4(b) muestra ADVERTISE ejecutándose en una configuración mínima de tres nodos tras el proceso de arranque.

4.3. Recuperación ante fallos de nodos

Con el fin de implementar mecanismos de contingencia para hacer frente a fallos de nodos, tuvimos que decidir qué propiedad queríamos preservar en ADVERTISE ante presencia de estas contingencias: la consistencia o la disponibilidad.

En nuestro caso de estudio, se decidió que la consistencia de datos era más importante, es decir, que no podía permitirse que campañas de publicidad, reglas o información de los medios se corrompiese o se perdiese en situaciones de inestabilidad. Las campañas de publicidad son productos por los que los anunciantes pagan, y el aspecto más importante del proceso publicitario es por tanto garantizar que un anuncio se muestran el número de veces, a los perfiles de clientes, y durante los intervalos de tiempo acordados.

Por el contrario, sí era aceptable que no se enviasen anuncios a los STB si en un determinado momento el sistema no estaba disponible para analizar las campañas y ejecutar las reglas vigentes para producir las acciones correspondientes. Dado que los anuncios se muestran a los clientes cuando navegan por los menús de contenidos del operador (se muestran sobreponiéndose al vídeo que se reproduce de fondo), en caso de no enviarse ninguna publicidad, simplemente el área de pantalla reservada para ese fin deja ver el canal que se encuentra sintonizado.

Además, dado que cada STB está conectado mediante una conexión no fiable de *long polling*, también se consideraba aceptable que un determinado anuncio se mostrase con algo de retraso en relación al instante temporal configurado en la campaña, especialmente si la contrapartida era evitar a toda cosa la corrupción de datos en ADVERTISE. Claramente, las consecuencias de la transmisión de datos no adecuados a los STB se consideraba un escenario mucho peor.

4.3.1. Comprobación de la integridad de los nodos

A pesar de haber elegido priorizar la consistencia a la disponibilidad, el sistema ADVERTISE está diseñado para ser tolerante al fallo de nodos, y aunque su funcionamiento pueda degradarse, continuará ejecutándose mientras los nodos se recuperan. Existe, con todo, una limitación estricta: para mantener el sistema en funcionamiento, debe haber un mínimo de dos nodos vivos conectados entre sí.

Como ya hemos explicado, cuando un nodo falla, los restantes se organizan automáticamente para tomar el control. Cada nodo ejecuta el mismo mecanismo de contingencia para manejar estas situaciones, comprobando la población de nodos activos de su clúster cada 10 segundos. Cuando la población de nodos activos se reduce a un sólo nodo, se suspende la ejecución de ADVERTISE, se monitoriza la conectividad, y se reinicia el nodo (para que se ejecuten los mecanismos de sincronización) una vez el clúster recupera dicha conectividad.

4.4. Recuperación de cortes en la red

Si un nodo pierde su conectividad, y es incapar de alcanzar al resto de nodos del clúster ADVERTISE, debemos controlar que no crea que son todos los demás nodos a excepción de él los que han caído y decida asumir todas las funciones del sistema, mientras que al mismo tiempo el resto del clúster siga funcionando con normalidad (o lanzando de nuevo la aplicación, si el aislado es el nodo maestro) en espera de que dicho nodo se recupere. De lo contrario, cuando la red se restablezca, podríamos encontrarnos con inconsistencias entre el conjunto de datos manejados por el nodo aislado frente al manejado por el resto de los nodos. También pueden darse otro tipo de inconsistencias, tales como la duplicación de responsabilidades (podríamos encontrarnos con dos nodos actuando de maestro). Por ello, decidimos que todo nodo que se encuentre solo suspenda inmediatamente su ejecución hasta que se restaure su conectividad. En ese momento, se reiniciará para poder resincronizar sus datos con los del resto del clúster.

Este comportamiento tiene el efecto secundario de que si fallan todos los nodos del sistema excepto uno, el nodo restante también se protegerá de la manera anterior y dejará de funcionar hasta que al menos uno de los otros nodos se haya recuperado. Ésta es la razón por la que la configuración mínima recomendada de ADVERTISE es la de un clúster de al menos tres nodos.

Hemos de señalar que esta versión de los mecanismos de contingencia de ADVERTISE no es capaz de detectar particiones de la red en las que un grupo de nodos quede aislado de otro grupo de nodos. Aunque cuando los nodos físicos están en el mismo segmento de red esta situación debería ser muy poco probable, si esto ocurre, el sistema tendría que ser reiniciado manualmente para forzar el proceso de inicialización y la sincronización de datos entre los dos clústeres.

4.4.1. Control de integridad de la aplicación distribuida

El mecanismo que verifica la conectividad entre los nodos se dedica fundamentalmente a enviar `pings` a los nodos del clúster cada diez segundos. Se asume que los nodos del clúster que no respondan están caídos. Si estaban en la lista de nodos activos hasta ese momento, se monitorizan al cabo de diez segundos. Si entonces siguen sin responder, se confirman como caídos. Cuando un nodo que estaba caído responde al `ping`, se le reinicia para forzar su sincronización de datos y que pueda volver a estar completamente operativo.

5. Conclusiones

El diseño de una aplicación distribuida requiere un análisis completo de sus necesidades, incluyendo todos los aspectos relacionados con su naturaleza concurrente y distribuida. Dependiendo del propósito u objetivo concreto de la aplicación, cada una de las propiedades clave del teorema CAP (consitencia, disponibilidad, tolerancia a particiones de red) tendrá diferente relevancia, lo que determinará un orden de prioridad entre ellas.

En este artículo hemos presentado una descripción detallada de las decisiones de diseño tomadas para que nuestro caso de estudio tuviese los mecanismos de tolerancia a fallos acordes con sus requisitos y su entorno de despliegue. En el desarrollo de dicho caso de estudio, el sistema distribuido ADVERTISE, construido para una operadora de cable con un número creciente de clientes (a día de hoy, alrededor de 100.000), debimos enfrentarnos con problemas habituales en los sistemas distribuidos como la caída de nodos o los cortes de red. En este trabajo, hemos explicado las aproximaciones que se siguieron para minimizar el impacto de dichos problemas en la consistencia, la disponibilidad y el rendimiento del sistema.

Fue el propio despliegue preliminar de ADVERTISE en su entorno de producción el que reveló ciertas casuísticas que no se habían tenido en cuenta durante el desarrollo inicial: la tendencia de ciertos nodos a fallar más que otros y la alta frecuencia de los cortes de red en determinados períodos temporales (*prime time*). Para conseguir que el sistema cumpliese sus requisitos de tolerancia a fallos, se diseñaron e incluyeron mecanismos de contingencia, que se volvieron esenciales para asegurar el correcto funcionamiento del sistema. En este caso, la consistencia fue la propiedad CAP considerada más importante para ADVERTISE, debido a la naturaleza del negocio publicitario y las necesidades de los anunciantes. Por este motivo, el sistema en su configuración actual no permite la ejecución en nodos aislados, sacrificando la disponibilidad en esas situaciones para poder evitar potenciales conflictos.

Si los requisitos de ADVERTISE hubiesen sido diferentes es posible que, por ejemplo, la disponibilidad hubiese resultado más importante que la consistencia. En cualquier caso, Erlang demostró ser una gran herramienta para enfrentarse con todas estas particularidades propias de los sistemas distribuidos, no sólo por la potencia de poder utilizar un lenguaje de programación funcional para implementar declarativamente comportamientos complejos, sino por las propias características del lenguaje y el modelo de distribución y concurrencia que implementa. No tenemos dudas sobre el hecho de que escoger Erlang y sus bibliotecas OTP como plataforma de desarrollo ha sido un factor clave en el éxito final del proyecto ADVERTISE.

Nuestras contribuciones en este artículo son:

- La descripción detallada de la arquitectura y funcionamiento general de ADVERTISE, un sistema distribuido de publicidad para una red de televisión digital, implementado usando Erlang/OTP, actualmente en funcionamiento y soportando cientos de miles de clientes.

- La descripción y discusión de las decisiones de diseño tomadas para asegurar el correcto funcionamiento de ADVERTISE después de su paso a producción, con consideración y estudio de los supuestos que Erlang hace sobre los sistemas distribuidos.

- La abstracción del problema y reflexión en líneas generales sobre los elementos decisivos a tener en cuenta para proteger de manera efectiva un sistema distribuido contra la caída de nodos y los cortes de red.

5.1. Trabajos futuros

Para aumentar aún más la confianza en el correcto funcionamiento del sistema ADVERTISE, nuestro trabajo a día de hoy se enfoca en la verificación de los módulos más críticos de ADVERTISE, tales como el motor de reglas, usando técnicas avanzadas como la prueba basada en modelos y en propiedades. También, debido al aspecto crítico de los mecanismos de *takeover* y *failover*, consideramos muy interesante el poder verificarlos utilizando herramientas de *model checking* como McErlang, siguiendo los pasos de [CGB+11].

6. Agradecimientos

Estamos eternamente agradecidos a Víctor M. Gulías por su ejemplo imperecedero, su apoyo incondicional y su incuestionable sabiduría. Sus incontables virtudes le sobreviven, y nos servirán de inspiración para siempre.

También queremos agradecer a Javier Mosquera su excelente trabajo en el proyecto ADVERTISE.

Referencias

[Arm07] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic programmers. Pragmatic Bookshelf, 2007.

[BBMS08] M. Balazinska, H. Balakrishnan, S. Madden, M. Stonebraker. Fault-tolerance in the borealis distributed stream processing system. *ACM Transactions on Database Systems* 33(1):3:1–3:44, Mar. 2008.

[Bre00] E. Brewer. Towards robust distributed systems. In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*. Pp. 7–10. 2000.

[CGB+11] D. Castro, V. Gulias, C. Benac-Earle, L. Fredlund, S. Rivas. A Case Study on Verifying a Supervisor Component Using McErlang. *Electronic Notes in Theoretical Computer Science* 271:23 – 40, 2011.

[CPV97] *Designing distributed applications with mobile code paradigms*. 1997.

[Cro06] D. Crockford. The application/json Media Type for JavaScript Object Notation (JSON). http://www.ietf.org/rfc/rfc4627, 2006.

[CT09] F. Cesarini, S. Thompson. *Erlang Programming*. O'Reilly Series. O'Reilly, 2009.

[DR08] T. Dierks, E. Rescorla. The Transport Layer Security (TLS) Protocol. http://tools.ietf.org/html/rfc5246, 2008.

[Erla] Erlang. Distribution protocol. http://www.erlang.org/doc/apps/erts/erl_dist_protocol.html.

[Erlb] Erlang. Heartbeat Monitoring of an Erlang Runtime System. http://www.erlang.org/doc/man/heart.html.

[GL02] S. Gilbert, N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News* 33(2):51–59, June 2002.

[Gof03] M. Goff. *Network Distributed Computing: Fitscapes and Fallacies*. Prentice Hall Professional Technical Reference, 2003.

[KDK+89] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, R. Zainlinger. Distributed fault-tolerant real-time systems: The Mars approach. *IEEE Micro* 9(1):25–40, 1989.

[LMC10] M. Logan, E. Merritt, R. Carlsson. *Erlang and OTP in action*. Manning Publications Co., 2010.

[MNW99] H. Mattsson, H. Nilsson, C. Wikstrom. Mnesia A Distributed Robust DBMS for Telecommunications Applications. *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, pp. 152–163, 1999.

[NT93] M. Norman, P. Thanisch. Models of machines and computation for mapping in multicomputers. *ACM Computing Surveys* 25(3):263–302, 1993.

[PZL08] *RESTful web services vs. "Big"web services: Making the right architectural decision*. 2008.

[SHB04] M. Shah, J. Hellerstein, E. Brewer. Highly available, fault-tolerant, parallel dataflows. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. SIGMOD '04, pp. 827–838. ACM, 2004.

[SJ05] A. Srinivas, D. Janakiram. A model for characterizing the scalability of distributed systems. *SIGOPS Operating Systems Review* 39(3):64–71, July 2005.

[Tho79] R. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems* 4(2):180–209, 1979.

[TLLS09] L. Tang, J. Li, Y. L, S. Shenker. An investigation of the Internet's IP-layer connectivity. *Computer Communications* 32(5):913–926, 2009.

[WJ99] M. Wooldridge, N. Jennings. Software engineering with agents: Pitfalls and pratfalls. *IEEE Internet Computing* 3(3):p 6, 1999.

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): Actas de las "XII Jornadas sobre Programación y Lenguajes (PROLE'2012), Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

Sesión 3

Tutorial y ponencia invitada

Sesión 3: Tutorial y ponencia invitada

Ricardo Peña. *Programación con paralelismo de datos en Haskell*.

Diana Parra. *Erlang, del laboratorio a la empresa*.

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): Actas de las "XII Jornadas sobre Programación y Lenguajes (PROLE'2012), Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

Programación con paralelismo de datos en Haskell (Tutorial)

Ricardo Peña[*]

ricardo@sip.ucm.es
Universidad Complutense de Madrid
Spain

Abstract: El paralelismo de datos es un paradigma dentro de la programación paralela caracterizado por ejecutar en paralelo la misma tarea sobre un subconjunto de los datos, los cuales suelen consistir en enormes vectores y matrices numéricas. El lenguaje de referencia es *High Performance Fortran*.

Resulta sorprendente conocer que ese paradigma puede practicarse en un lenguaje funcional como Haskell. La librería *Data Parallel Haskell* proporciona una interfaz de alto nivel que permite a los programadores escribir sus programas con las funciones de orden superior habituales sobre listas (*map, foldr, zip, scan, enumerate,* ...), y sin embargo conseguir que dichos programas se ejecuten en paralelo en arquitecturas multi-núcleo. La clave está en la gran cantidad de transformaciones que sufren los programas durante la compilación.

Este tutorial describe los detalles de dichas transformaciones y muestra que es compatible una programación de alto nivel con una ejecución eficiente en una arquitectura paralela.

Keywords: Paralelismo, Haskell.

[*] Trabajo con financiación parcial de proyectos TIN2008-06622-C03-01/TIN (STAMP) y S2009/TIC-1465 (PROMETIDOS).

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): PROLE'2012, pp. 295, ISBN:978-84-15487-27-2.
Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): Actas de las "XII Jornadas sobre Programación y Lenguajes (PROLE'2012), Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

Erlang, del laboratorio a la empresa (Charla invitada)

Diana Parra Corbacho

diana.corbacho@erlang-solutions.com
http://www.erlang-solutions.com/
Erlang Solutions Ltd
29 London Fruit & Wool Exchange
Brushfield Street
London, E1 6EU, UK

Abstract: En los últimos años la explosión tecnológica ha dado lugar a nuevas formas de comunicación, diversión y negocios. Redes sociales, juegos online, dispositivos embebidos, sistemas bancarios... ¿Qué tienen en común todos ellos? Concurrencia, distribución, tolerancia a fallos o escalabilidad son algunas de sus principales características.

Desde que Erlang fue liberado como código abierto por primera vez en 1998, el lenguaje funcional desarrollado en los laboratorios de Ericsson se ha convertido en la solución a muchos de estos problemas. Protocolos de señalización como SIP; comunicación con XMPP para sistemas de geolocalización, de control, mensajes instantáneos o juegos; SMS gateways; bases de datos NoSQL; switchs de transferencias bancarias en tiempo real o virtualización para computación de alto rendimiento son algunos de los proyectos empresariales desarrollados en Erlang.

Keywords: Erlang, aplicaciones, sistemas distribuidos, computación paralela.

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): PROLE'2012, pp. 297, ISBN:978-84-15487-27-2.
Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): Actas de las "XII Jornadas sobre Programación y Lenguajes (PROLE'2012), Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

M.M. Gallardo, M. Villaret, L. Iribarne (Eds.): Actas de las "XII Jornadas sobre Programación y Lenguajes (PROLE'2012), Jornadas SISTEDES'2012, Almería 17-19 sept. 2012, Universidad de Almería.

Sistedes 2012
#sistedes2012 Almería

JISBD PROLE JCIS

ACG
Applied Computing Group
Ref. TIC-211, Junta de Andalucía
University of Almería, Spain
Ctra. Sacramento s/n, 04120 Almería

FUNDACIÓN MEDITERRÁNEA
EMPRESA - UNIVERSIDAD DE ALMERIA

Colaboradores

GOBIERNO DE ESPAÑA
MINISTERIO DE ECONOMÍA Y COMPETITIVIDAD

JUNTA DE ANDALUCIA
CONSEJERÍA DE ECONOMÍA, INNOVACIÓN Y CIENCIA

DIPUTACIÓN DE ALMERÍA

AYUNTAMIENTO DE ALMERÍA

ALBORADA Ingeniería I+D+I

Escuela Superior de Ingeniería
UNIVERSIDAD DE ALMERÍA

UNIVERSIDAD DE ALMERÍA
Vic. de Tecnologías de la Información y de la Comunicación
Vic. de Investigación, Desarrollo e Innovación
Unidad de Enseñanza Virtual del VTIC (EVA/TIC)
Departamento de Lenguajes y Computación

UNIVERSIDAD DE ALMERÍA
Plan Propio de Investigación

cajamar

PITA Parque Científico-Tecnológico de Almería

IEEE computer society
IEEE Computer Society – Spanish Chapter

ices Ingeniería, Consultoría y Estrategia de Sistemas

almerimatik i+d+i

http://sistedes2012.ual.es