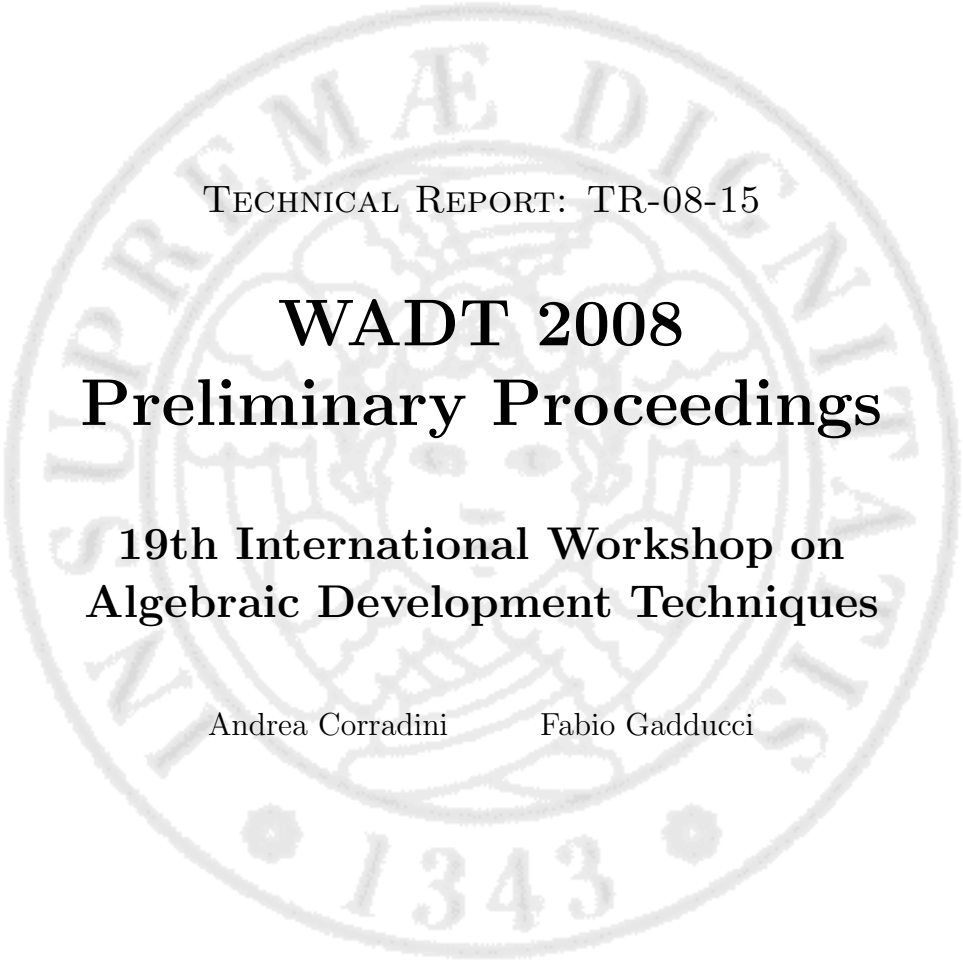
Università di Pisa
Dipartimento di Informatica



Technical Report: TR-08-15

WADT 2008
Preliminary Proceedings

19th International Workshop on Algebraic Development Techniques

Andrea Corradini Fabio Gadducci

June 9, 2008

address: Largo B. Pontecorvo 3, 56127 Pisa, Italy. tel: +39 050 2212700 fax: +39 050 2212726

Preface

After having joined forces with the International Workshop on Coalgebraic Methods in Computer Science (CMCS) for the Second International Conference on Algebra and Coalgebra in Computer Science (CALCO 2007), the Nineteenth International Workshop on Algebraic Development Techniques (WADT 2008) was held as an individual workshop and in its traditional form in Pisa, Italy, from the 13th to the 16th of June 2008.

The algebraic approach to system specification encompasses many aspects of the formal design of software systems. Originally born as a formal method for reasoning about abstract data types, it now covers new specification frameworks and programming paradigms (such as object-oriented, aspect-oriented, agent-oriented, logic and higher-order functional programming) as well as a wide range of application areas (including information systems and concurrent, distributed and mobile systems).

The WADT workshop series aims to provide a platform for presenting recent and ongoing work, to meet colleagues, and to discuss new ideas and future trends. Typical, but not exclusive topics of interest are

- foundations of algebraic specification
- process calculi and models of concurrent, distributed and mobile computing
- specification languages, methods, and environments
- semantics of conceptual modelling methods and techniques
- model-driven development
- graph transformations, term rewriting and proof systems
- integration of formal specification techniques
- formal testing and quality assurance
- validation and verification

This report contains the thirtythree abstracts presented during the workshop: they were selected by the Steering Committee on the basis of the submitted abstracts according to originality, significance, and general interest. In addition to the presentations of ongoing research results, the programme included three invited lectures by Egon Bœrger (Dipartimento di Informatica, Pisa), Luca Cardelli (Microsoft Research, Cambridge) and Stephen Gilmore (Laboratory for Foundations of Computer Science, Edinburgh).

As for previous WADT workshops, after the meeting selected authors will be invited to submit full papers for the refereed proceedings, which will be published as a volume of Lecture Notes in Computer Science (Springer Verlag).

The Steering Committee included

- Michel Bidoit (France)
- José Fiadeiro (chair, UK)
- Hans-Jörg Kreowski (Germany)
- Till Mossakowski (Germany)
- Peter Mosses (UK)
- Fernando Orejas (Spain)
- Francesco Parisi-Presicce (Italy and USA)
- Andrzej Tarlecki (Poland)

The local Organising Committee included also Filippo Bonchi, Roberto Bruni, Vincenzo Ciancia, Andrea Corradini (chair) and Fabio Gadducci.

The workshop took place under the auspices of IFIP WG 1.3, and it was organized by the Department of Informatics of Pisa University. We gratefully acknowledge the sponsorship by IFIP TC1 and by the University of Pisa.

June 2008 Andrea Corradini and Fabio Gadducci

Table of Contents

June 14, morning

June 14, afternoon

June 15, morning

June 16, morning

June 16, afternoon

Semantics of business process modelling notations

Egon Börger

Dipartimento di Informatica, Pisa, Italia
boerger@di.unipi.it

Abstract

We use Abstract State Machines to develop an extensible semantical framework for business process modeling notations. The approach is illustrated by defining a high-level interpreter for business process diagrams written in the OMG standard BPMN. We show, by presenting various solutions of the so-called OR-join problem, how ASM models for BPMN diagrams can be used for their accurate analysis.

This is joint work with B. Thalheim and part of the Humboldt Research Award project, hosted by the Chair for Information Systems Engineering at the Computer Science Department of the University of Kiel/Germany.

Molecules as Automata

Luca Cardelli

Microsoft Research, Cambridge, U.K.
luca@microsoft.com

Molecular biology investigates the structure and function of biochemical systems starting from their basic building blocks: macromolecules. A macromolecule is a large, complex molecule (a protein or a nucleic acid) that usually has inner mutable state and external activity. Informal explanations of biochemical events trace individual macromolecules through their state changes and their interaction histories: a macromolecule is endowed with an identity that is retained through its transformations, even through changes in molecular energy and mass. A macromolecule, therefore, is qualitatively different from the small molecules of inorganic chemistry. Such molecules are stateless: in the standard notation for chemical reactions they are seemingly created and destroyed, and their atomic structure is used mainly for the bookkeeping required by the conservation of mass.

Attributing identity and state transitions to molecules provides more than just a different way of looking at a chemical event: it solves a fundamental difficulty with chemical-style descriptions. Each macromolecule can have a huge number of internal states, exponentially with respect to its size, and can join with other macromolecules to from even larger state configurations, corresponding to the product of their states. If each molecular state is to be represented as a stateless chemical species, transformed by chemical reactions, then we have a huge explosion in the number of species and reactions with respect to the number of different macromolecules that actually, physically, exist. Moreover, macromolecules can join to each other indefinitely, resulting in situations corresponding to infinite sets of chemical reactions among infinite sets of different chemical species. In contrast, the description of a biochemical system at the level of macromolecular states and transitions remains finite: the unbounded complexity of the system is implicit in the potential molecular interactions, but does not have to be written down explicitly. Molecular biology textbooks widely adopt this finite description style, at least for the purpose of illustration.

Many proposal now exist that aim to formalize the combinatorial complexity of biological systems without a corresponding explosion in the notation. Macromolecules, in particular, are seen as stateful concurrent agents that interact with each other through a dynamic interface. While this style of descriptions is (like many others) not quite accurate at the atomic level, it forms the basis of a formalized and growing body of biological knowledge.

The complex chemical structure of a macromolecule is thus commonly abstracted into just internal states and potential interactions with the environment. Each macromolecule forms, symmetrically, part of the environment for the other macromolecules, and can be described without having to describe the whole

environment. Such an open system descriptive style allows modelers to extend systems by composition, and is fundamental to avoid enumerating the whole combinatorial state of the system (as one ends up doing in closed systems of chemical reactions). The programs-as-models approach is growing in popularity with the growing modeling ambitions in systems biology, and is, incidentally, the same approach taken in the organization of software systems. The basic problem and the basic solution are similar: programs are finite and compact models of potentially unbounded state spaces.

At the core, we can therefore regard a macromolecule as some kind of automaton, characterized by a set of internal states and a set of discrete transitions between states driven by external interactions. We can thus try to handle molecular automata by some branch of automata theory and its outgrowths: cellular automata, Petri nets, and process algebra. The peculiarities of biochemistry, however, are such that until recently one could not easily pick a suitable piece of automata theory off the shelf. Many sophisticated approaches have now been developed, and we are particularly fond of stochastic process algebra. In this talk, however, we do our outmost to remain within the bounds of a much simpler theory. We go back, in a sense, to a time before cellular automata, Petri nets and process algebra, which all arose from the basic intuition that automata should interact with each other. Our main criterion is that, as in finite-state automata, we should be able to easily and separately draw the individual automata, both as a visual aid to design and analysis, and to emulate the illustration-based approach found in molecular biology textbooks.

With those aims, we investigate stochastic automata collectives. Technically, we place ourselves within a small fragment of a well-know process algebra (stochastic pi-calculus), but the novelty of the application domain, namely the mass action behavior of large numbers of well-mixed automata, demands a broader outlook. By a collective we mean a large set of interacting, finite state automata. This is not quite the situation we have in classical automata theory, because we are interested automata interactions. It is also not quite the situation with cellular automata, because our automata are interacting, but not necessarily on a regular grid. And it is not quite the situation in process algebra, because we are interested in the behavior of collectives, not of individuals. And in contrast to Petri nets, we model separate parts of a system separately. By stochastic we mean that automata interactions have rates. These rates induce a quantitative semantics for the behavior of collectives, and allow them to mimic chemical kinetics. Chemical systems are, physically, formed by the stochastic interactions of discrete particles. For large number of particles it is usually possible to consider them as formed by continuous quantities that evolve according to deterministic laws, and to analyze them by ordinary differential equations. However, one should keep in mind that continuity is an abstraction, and that sometimes it is not even a correct limit approxima-tion.

In biochemistry, the stochastic discrete approach is particularly appropriate because cells often contain very low numbers of molecules of critical species: that is a situation where continuous models may be misleading. Stochastic au-

tomata collectives are hence directly inspired by biochemical systems, which are sets of interacting macromolecules, whose stochastic behavior ultimately derives from molecular dynamics. Some examples of the mismatch between discrete and continuous models are discussed.

Service-Level Agreements for Service-Oriented Computing

Allan Clark, Stephen Gilmore and Mirco Tribastone

Laboratory for Foundations of Computer Science
The University of Edinburgh, Scotland

Abstract. Service-oriented computing is dynamic. There may be many possible service instances available for binding, leading to uncertainty about where service requests will execute. We present a novel Markovian process calculus which allows the formal expression of uncertainty about binding as found in service-oriented computing. We show how to compute meaningful quantitative information about the quality of service provided in such a setting. These numerical results can be used to allow the expression of accurate service-level agreements about service-oriented computing.

1 Introduction

Analytical or numerical performance evaluation provides valuable insights into the timed behaviour of systems over the short or long run. Prominent methods used in the field include the numerical evaluation of continuous-time Markov chains (CTMCs). These bring a controlled degree of randomness to the system description by using exponentially-distributed random variables governed by rate constants to characterise activities of varying duration. Often generated from a high-level description language such as a Petri net or a process algebra, CTMCs are applied to study fixed, static system configurations with known subcomponents with known rate parameters. This is far from the operating conditions of service-oriented computing where for critical service components a set of replacements with perhaps vastly different performance qualities stand ready to substitute for components which are either unavailable, or the consumer just simply chooses not to bind to them. How can we bridge this gap and apply Markovian performance evaluation to the assessment of service-level agreements about service-oriented computing?

SRMC is a Markovian process calculus in the tradition of PEPA [1], Stochastic KLAIM [2], and Stochastic FSP [3]. On top of a classical process calculus, SRMC adds *namespaces* to allow the structured description of models of large size, and *dynamic binding* to represent uncertainty about component specification or the values of parameters. As a first step in machine processing, namespaces and dynamic binding can be resolved in order to map into a Markovian calculus without these features such as PEPA (for performance analysis [4, 5]).

2 Example: Distributed e-Learning Case Study

The SVU is a virtual organisation formed by bringing together the resources of the universities at Edinburgh (UEDIN), Munich (LMU), Bologna (UNIBO), Pisa (UNIPI) and others not listed in this example. The SVU federates the teaching and assessment capabilities of the universities allowing students to enrol in courses irrespective of where they are delivered geographically. Students download *learning objects* from the content download portals of the universities involved and upload archives of their project work for assessment. By agreement within the SVU, students may download from (or upload to) the portals at any of the SVU sites, not just the one which is geographically closest.

The portals at Edinburgh are described in SRMC thus.

```
UEDIN::{
   lambda = 1.65;  mu = 0.0275;  gamma = 0.125;  delta = 3.215;
   avail = { 0.7, 0.8, 0.9, 1.0 };
   UploadPortal::{
      Idle = (upload, avail * lambda).Idle + (fail, mu).Down;
      Down = (repair, gamma).Idle;
   }
   DownloadPortal::{
      Idle = (download, avail * delta).Idle + (fail, mu).Down;
      Down = (repair, gamma).Idle;
   }
}
```

The portals at Munich are so reliable that it is not worth modelling the very unlikely event of their failure. However, they are slower than the equivalent portals at Edinburgh and availability is more variable and usually lower, because the portals are serving a larger pool of local students. Because it is running a more recent release of the portal software the Bologna site offers secure upload and download also. Availability is usually very good. To maintain good availability the more expensive operations of secure upload and secure download are not offered if the system seems to be becoming heavily loaded. The Pisa site is just like the Bologna site, but uses a higher grade of encryption, meaning that secure upload and download are slower (`slambda` = 0.975, `sdelta` = 1.765). We can list the possible bindings for upload and download portals in the following way.

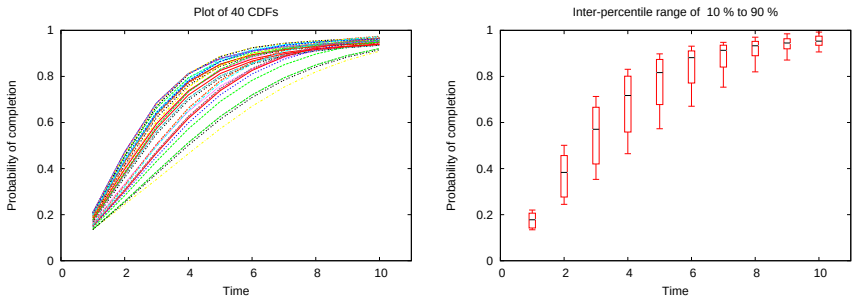
```
UploadPortal =
   { UEDIN::UploadPortal::Idle, LMU::UploadPortal::Idle,
     UNIBO::UploadPortal::Idle, UNIPI::UploadPortal::Idle };

DownloadPortal =
   { UEDIN::DownloadPortal::Idle, LMU::DownloadPortal::Idle,
     UNIBO::DownloadPortal::Idle, UNIPI::DownloadPortal::Idle };
```

In this example the clients of the system wish to download three sets of learning materials and to upload two coursework submissions. We are interested in the

passage time from the start to the finish of this activity. The complete system is formed by composing the client with the two portals, cooperating over upload and download. The upload and download portals do not communicate with each other (<>).

```
System = Client <upload, download, suploaad, sdownload>
              (UploadPortal <> DownloadPortal);
```



(a) (b)

Fig. 1. Sub-figure (a) shows 40 of the response-time distributions computed for the Sensoria Virtual University example. Sub-figure (b) shows the 10% to 90% percentile of the results over all of the runs. The median value is also marked as a horizontal line cutting across the thick bar in the candlestick. From sub-figure (b) we can report results of the form "All uploads and downloads will have completed by time $t = 10$ with probability between 0.90 and 0.97, in 90% of configurations".

References

1. Hillston, J.: A Compositional Approach to Performance Modelling. Cambridge University Press (1996)
2. De Nicola, R., Katoen, J.P., Latella, D., Massink, M.: STOKLAIM: A stochastic extension of KLAIM. Technical Report ISTI-2006-TR-01, Consiglio Nazionale delle Ricerche (2006)
3. Ayles, T.P., Field, A.J., Magee, J., Bennett, A.: Adding Performance Evaluation to the LTSA Tool. In: Tool demonstration, 13th International Conference on Computer Performance Evaluation: Modelling Techniques and Tools, September 2003. (September 2003)
4. Clark, A.: The ipclib PEPA Library. In Harchol-Balter, M., Kwiatkowska, M., Telek, M., eds.: Proceedings of the 4th International Conference on the Quantitative Evaluation of SysTems (QEST), IEEE (September 2007) 55–56
5. Tribastone, M.: The PEPA Plug-in Project. In Harchol-Balter, M., Kwiatkowska, M., Telek, M., eds.: Proceedings of the 4th International Conference on the Quantitative Evaluation of SysTems (QEST), IEEE (September 2007) 53–54

Observability concepts in abstract data type specification, 30 years later*

Donald Sannella[1] and Andrzej Tarlecki[2,3]

[1] Laboratory for Foundations of Computer Science, University of Edinburgh
[2] Institute of Informatics, Warsaw University
[3] Institute of Computer Science, Polish Academy of Sciences

The starting point for this work is a brief paper [1], which appears to be the first of many papers to study observational aspects of the algebraic approach to software specification and development, where the overall idea is that one should regard a specification of a system as constraining its observable behaviour, and nothing more. Such a view is required to cope with many examples. However, it adds significant technical complexities to the simple and elegant algebraic approach. Some of these remain unresolved today, even after 30 years of research.

[1] starts by challenging the initial algebra approach to specifications of abstract data types, then recently introduced by early versions of [2]. Most importantly, [1] points out that not all sorts of data in a data type play the same role: one should separate the given, "old" sorts from the "new" ones, to be specified and implemented. What really matters then is the behaviour of the data type as viewed via these old sorts only; the implementation details of the new sorts play a secondary role. Such *observable behaviour* is captured by the evaluation function restricted to terms that are of old sorts, but in general use the new operations and involve new sorts internally. Another crucial insight in [1] is that in general there are many non-isomorphic algebras that display the same observable behaviour. They show that the set of isomorphism classes of such algebras (limited to the ones generated by the old sorts) forms a complete lattice — a nice technical result which, however, is not used to insist that any such specific algebra is always chosen since all of them are equally adequate implementations of the given observable behaviour. Such behaviours are specified in [1] by giving a partial evaluation function, which assigns values to some terms of old sorts only, marking the others as "don't care" cases (indicated by assigning to them a special "value" α, a notation that we will maintain here). The latter captures the situation where the specifier permits the behaviour to be chosen arbitrarily (but consistently with other choices) in any particular implementation. Particular implementations for such a behaviour specification in [1] are captured as (generated) algebras that conform to the specification in the obvious sense.

Quite a few points made in [1] were very insightful in their historical context. This is the first place we know of where several key ideas appear, including some that underlie most of our own contributions to the area. First, the stress on the need for loose specifications, which need not determine behaviour unambiguously

* This work has been partially supported by European projects IST-2005-015905 MOBIUS (DS, AT) and IST-2005-016004 SENSORIA (AT).

(up to isomorphism) was of key importance. The results on the lattice properties of the class of models for a given observable behaviour initiated a line of research in this direction, including a debate on the issue of initial vs. final interpretation of algebraic specifications. One aspect which disappeared in later work was the method of presenting specifications by using an explicitly given set of data on which the data type is based, with behaviour specified by indicating the results of evaluation of some terms, while explicitly marking others as "don't care" cases. The main contribution though is the idea of limiting specifications to observable parts of behaviour only, thus introducing observability aspects to algebraic specification.

We reiterate some of the ideas presented in [1], looking back at more than 30 years of work on algebraic specification, and trying to blend what happened with these ideas with our current personal perspective. We sketch a framework for observable behaviour specification and development, reconsidering some of the work presented earlier in a different technical setting. It is reassuring that, after shifting to quite a different specification technology, inspired by [1], our basic ideas on system specification, architectural design and development under an observational view of specifications still stand.

[The corresponding paper in the Montanari Festschrift (Springer LNCS 5065, pages 593–617, 2008) is available from `http://homepages.inf.ed.ac.uk/dts/pub/montanari-festschrift.pdf`.]

References

1. Giarratana, V., Gimona, F., Montanari, U.: Observability concepts in abstract data type specifications. In: Proc. 1976 Symp. on Mathematical Foundations of Computer Science, Springer LNCS 45 (1976) 567–578
2. Goguen, J., Thatcher, J., Wagner, E.: An initial algebra approach to the specification, correctness and implementation of abstract data types. In: Current Trends in Programming Methodology, Vol. 4: Data Structuring. Prentice-Hall (1978) 80–149 Edited by R.T. Yeh.

What is a Multi-Modelling Language?*

Artur Boronat[1], Alexander Knapp[2], Jose Meseguer[3], and Martin Wirsing[2]

[1] University of Leicester
[2] Ludwig-Maximilians-Universität München
[3] University of Illinois at Urbana-Champaign

In an ideal software engineering world, development teams would follow well-defined processes in which one single modelling language is used for all requirements and design documents; but in practice ”multi-modelling” happens: in a large software project entity relationship diagrams and XML may be used for domain modelling, BPEL for business process orchestration, and UML for design and deployment. UML itself can be seen as a multi-modelling language comprising several sublanguages such as class diagrams, OCL and state machines; each submodelling language provides a particular view on a software system. Such views have the advantage of complexity reduction: a software engineer can concentrate on a particular aspect of the system such as the domain architecture or dynamic interactions between objects.

On the other hand, multi-modelling raises a number of methodological and semantical questions: are the different sublanguages semantically consistent, how can we correctly transform an abstract model of one modelling language into a more concrete one in another language? More generally, is there a notion of ”multi-modelling language” which provides more insight than just a bunch of modelling languages together? Is it possible to give a semantics to multi-modelling languages which allows one to deal with consistency, validation and verification but retain the advantages of views by providing a local semantics and local reasoning capabilities for each modelling language?

In the literature, there are three complementary approaches for interrelating modelling notations: the ”system model approach”, the ”model-driven architecture approach”, and the ”heterogeneous semantics and development approach”. In the system model approach the different modelling languages are translated into a common (formally defined) modelling notation called system model [1] which serves as unique semantic basis and for analysing consistency of software engineering models. In the ”model-driven architecture approach” [2] model transformations are used for semi-automatically transforming platform-independent models into platform-specific models; consistency questions are typically dealt with at the syntactic level of the modelling notation. In the third approach different modelling languages are interrelated by semantic-preserving mappings [3, 4]; a mathematical semantics is given locally for each modelling language and the consistency between different languages is analysed semantically through the semantic-preserving mappings. All three approaches have been applied to several modelling languages including UML, but to our knowledge, multi-modelling languages have never been systematically studied.

In this paper we combine ideas from model-driven architecture and heterogeneous semantics and propose a new, semantically well-founded notion of a multi-modelling language and a new notion of semantic correctness for model transformations.

* This work has been partially sponsored by the project SENSORIA IST-2005-016004

In particular, our formal definition of a multi-modelling language L

- uses the Meta-Object Facility MOF and their algebraic semantics [5] for describing the metamodels and models of the sublanguages of L
- associates an institution to each sublanguage S of L and a gives a mathematical semantics to each software engineering model [4] of S by a corresponding (logical) theory in the institution of S
- defines the links between different sublanguages of S by model transformations and provides a notion of semantic correctness for such transformations
- provides a notion of consistent heterogeneous (software engineering) model of the multi-modelling language L which is derived from a notion of a category of heterogeneous mathematical models at the institution level.

In the full paper we will illustrate these ideas in the context of existing modelling languages by presenting a case study which involves models in several modelling languages, and explain how our concepts can be applied to show the consistency of software engineering models and the semantic correctness of model transformations. In particular, we choose UML and entity relationship diagrams as modelling languages and combine them via a semantically correct model transformations to a multi-modelling language. Based on earlier work [4] we show that class diagrams and OCL form a multi-modelling language where class diagrams are related to OCL by a semantically correct model transformation. Then we obtain the full multimodelling language by a semantically correct model transformation from class diagrams to entity relationship diagrams.

References

1. Broy, M., Cengarle, M.V., Rumpe, B.: Semantics of UML – Towards a System Model for UML: The Structural Data Model. Technical Report TUM-I0612, Institut für Informatik, Technische Universität München (2006)
2. Object Management Group (OMG): MDA Guide Version 1.0.1. Technical report, OMG (2003) `www.omg.org/docs/omg/03-06-01.pdf`.
3. Mossakowski, T.: Heterogeneous Specification and the Heterogeneous Tool Set. Habilitation thesis, Universität Bremen (2005)
4. Cengarle, M.V., Knapp, A., Tarlecki, A., Wirsing, M.: A Heterogeneous Approach to UML Semantics. In: Festschrift for Ugo Montanari. Volume 5019 of Lect. Notes in Comp. Sci., Springer (2008) To appear.
5. Boronat, A., Meseguer, J.: An Algebraic Semantics for MOF. In: FASE 2008, Budapest, Hungary, March 29-April 6, Proceedings. Lect. Notes in Comp. Sci., Springer (2008)

[4] For distinguishing semantic models from the models of a modelling language we write ”software engineering model” for a (syntactic) description of a model in a modelling language such as UML. In contrast to this, ”(semantic) models” are part of the mathematical semantics of a modelling language and therefore a semantic model can be understood as a model of a theory in a suitable logic.

An institution for processes and data

Till Mossakowski[1] and Markus Roggenbach[2]

[1] DFKI Lab Bremen and University of Bremen, Germany,
Till.Mossakowski@dfki.de
[2] Swansea University, United Kingdom,
M.Roggenbach@Swan.ac.uk

CSP-CASL [1] is a comprehensive specification language which combines *processes* written in the process algebra CSP [2, 3] with the specification of *data types* formulated in algebraic specification language CASL [4]. Recent developments on CSP-CASL cover tool support [5] as well as testing from CSP-CASL specifications [6].

In this talk we address the question of how to formulate CSP-CASL as an institution [7]. The CSP-CASL semantics follows a two-step approach: in its first step, the data specified in CASL is turned into an alphabet of communications, in its second step, the CSP process semantics is applied. This allows us to base our new formulation of a CSP-CASL institution on our previous work concerning CSP alone [8].

Solving this fundamental question of semantic nature has impact on the specification practice: the institution independent structuring mechanisms of CASL [4] become available within CSP-CASL specifications; furthermore, as projecting from CSP-CASL institution into CASL institution yields an institution morphism [9], it is also possible to use CSP-CASL within heterogeneous specifications [10, 11].

References

1. Roggenbach, M.: CSP-CASL: A new integration of process algebra and algebraic specification. Theoretical Computer Science **354**(1) (2006) 42–71
2. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall (1985)
3. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice Hall (1998)
4. Mosses, P.D., ed.: CASL Reference Manual. LNCS 2960. Springer (2004)
5. O'Reilly, L., Isobe, Y., Roggenbach, M.: Integrating Theorem Proving for Processes and Data. In Haveraaen, M., Power, J., Seisenberger, M., eds.: CALCO-jnr 2007, University of Bergen (to appear)
6. Kahsai, T., Roggenbach, M., Schlingloff, B.H.: Specification-based testing for refinement. In Hinchey, M., Margaria, T., eds.: SEFM 2007, IEEE Computer Society (2007) 237–247
7. Goguen, J.A., Burstall, R.M.: Institutions: abstract model theory for specification and programming. J. ACM **39**(1) (1992) 95–146
8. Mossakowski, T., Roggenbach, M.: Structured CSP – A Process Algebra as an Institution. In Fiadeiro, J.L., Schobbens, P.Y., eds.: WADT 2006. LNCS 4409 (2007) 92–110
9. Goguen, J., Roşu, G.: Institution morphisms. Formal aspects of computing **13** (2002) 274–307

10. Mossakowski, T., Maeder, C., Lüttich, K.: The Heterogeneous Tool Set. In Grumberg, O., Huth, M., eds.: TACAS 2007. Volume 4424 of Lecture Notes in Computer Science., Springer-Verlag Heidelberg (2007) 519–522
11. Mossakowski, T.: Heterogeneous specification and the heterogeneous tool set. Technical report, Universitaet Bremen (2005) Habilitation thesis.

Refinement notions for CSP-CASL

Temsghen Kahsai and Markus Roggenbach

Swansea University, United Kingdom,
{csteme,csmarkus}@swan.ac.uk

In this talk we give a status report of an ongoing PhD project which develops and studies various notions of refinement for the specification language CSP-CASL [1]. CSP-CASL combines the description of *processes* written in the process algebra CSP [2, 3] with the specification of *data types* formulated in the algebraic specification language CASL [4].

The starting points for the PhD project are the various notions of refinement for CSP and CASL alone. For CSP, each of its semantic models induces a notion of refinement, i.e., the model $\mathcal{T}$ induces the notion of trace refinement which preserves safety properties, the model $\mathcal{N}$ induces the notion of failures-divergence refinement which preserves lifelock-freedom, and the model $\mathcal{F}$ induces the notion of stable-failures refinement which preserves deadlock-freedom. In algebraic specification [5], on the other side, we have model class inclusion as the simplest form of refinement, while, for instance, observational refinement [6] captures a more 'refined' relation between model classes.

In our project, we combine refinement notions on CSP and CASL alone into refinement notions for CSP-CASL. Having CSP-CASL available as an institution, every such CSP-CASL refinement for basic specifications can also be seen as a refinement that allows one to change the signature. The case study on the electronic payment system EP2 [7] yields good practical insight into the question if such a newly designed refinement notion is useful. On the theoretical side, we study decomposition theorems and the question if properties such as deadlock-freedom are preserved under CSP-CASL refinement.

References

1. Roggenbach, M.: CSP-CASL – A new integration of process algebra and algebraic specification. Theoretical Computer Science **354** (2006) 42–71
2. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall (1985)
3. Roscoe, A.: The theory and practice of concurrency. Prentice Hall (1998)
4. Astesiano, E., Bidoit, M., Kirchner, H., Krieg-Brückner, B., Mosses, P.D., Sannella, D., Tarlecki, A.: CASL: the common algebraic specification language. Theoretical Computer Science **286**(2) (September 2002) 153–196
5. E. Astesiano, H.-J. Kreowski, B.B.: Algebraic Foundations of Systems Specifications. Springer (1999)
6. Bidoit, M., Hennicker, R.: Constructor-based observational logic. Journal of Logic and Algebraic Programming **67**(1-2) (April-May 2006) 3–51
7. Gimblett, A., Roggenbach, M., Schlingloff, H.: Towards a formal specification of an electronic payment systems in CSP-CASL. In: Revised Selected Papers of WADT'04. LNCS 3423. Springer (2005)

Towards a Spatial Temporal Logic for Graph Transformation

Andrea Corradini[1] and Reiko Heckel[2]

[1] Dipartimento di Informatica, Universitá di Pisa
andrea@di.unipi.it
[2] Department of Computer Science, University of Leicester
reiko@mcs.le.ac.uk

In the past decade the verification of concurrent and distributed systems modelled by graph transformation has been receiving increased attention [1, 2]. Approaches range from theorem proving, via model checking, to testing. In particular, approaches based on model checking use various types of temporal logics to express behavioural properties of graph transformation systems. The design of such a logic has to take account of the models chosen to represent systems, i.e., the connectives and primitives of the logic will depend on the relevant structure of models, its expressivity has to be balanced with the chosen semantic equivalence, etc. For example, different logics are usually required for models based on transition systems up to bisimulation, sets of traces or partial orders. At the same time, the structure of states (e.g., whether they are given by sets, multisets, trees, or graphs) could be reflected in the logic by means of connectives navigating that structure. A good understanding of the relevant model of computation is therefore essential.

Algebraic and categorical techniques provide powerful tools for analysing the structure of such models. A general method for the development of models of computations from given formal models of programs (called *Structured Transition Systems (STS)*) has been introduced in [3]. Models of computations are there given by categories with algebraic structure on objects and morphisms. The categorical composition operation models the sequential composition of computations, whereas the algebraic structure is used to express the distribution of states and computations. In order to apply this method to a particular domain, only the algebraic structure of the states has to be given. Then, the construction of computational models from given atomic state transitions (rules) is fully determined by this structure. Thus the main problem is to find the right structure of the states in order to obtain the appropriate notion of computation.

Using this general method, in this paper a computational model of graph transformation is developed which extends the classical model [4] by a second dimension modelling the distributed structure of states and computations.

A model of computation for a graph transformation system in [4] is a category whose objects are graphs and whose arrows are equivalence classes of (global) graph derivations with respect to the shift-equivalence. The corresponding structure of sequential composition (in time) is referred to as *vertical* structure. The additional *horizontal* structure describes the composition (in space) of graphs and derivations by pushouts (or, more generally, finite colimits), that is, the

gluing of local graphs along common interfaces, and the corresponding construction of more global derivations from local ones. Abstractly speaking, we obtain a double category with finite horizontal colimits, a structure closely related to the Tile Model [5] but for the fact that in our model object rather than arrows of a category represent the states.

The model of computation developed in this paper is a distributed presentation of the classical model. In fact, disregarding the distribution structure, the classical model of [4] is obtained from our model as a projection [6].

Next we define a temporal logic for concurrent computations in a graph transformation system. The logic represents a specialisation of van Benthem's arrow logic to categories, extended by spatial connectives like a concurrent composition of computations, which reflect the two-dimensional structure of the computational model.

It includes basic propositions, like *state patterns* and *rules* and provides *temporal operators*, including pre- and post-conditions, sequential composition and idle computations, an operator for *spatial composition* corresponding to horizontal pushouts, and the usual connectives and quantifiers of first order logic. Formulas either express properties of states or of computations.

Besides presenting the definition of the logic, we discuss its semantics, deduction rules, and potential applications to other transformation systems on diagrams and high-level structures.

References

1. Rensink, A., Heckel, R., König, B., eds.: Proceedings of the Workshop on Graph Transformation for Verification and Concurrency (GT-VC 2005), San Francisco, CA, USA. Volume 154(2) of Electronic Notes in TCS., Elsevier Science (2006)
2. Rensink, A., Heckel, R., König, B., eds.: Proceedings of the Workshop on Graph Transformation for Concurrency and Verification (GT-VC 2006), Bonn, Germany. Volume 175 of Electronic Notes in TCS., Elsevier Science (2007)
3. Corradini, A., Montanari, U.: An algebraic semantics for structured transition systems and its application to logic programs. Theoret. Comput. Sci. **103** (1992) 51–106
4. Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R., Löwe, M.: Algebraic approaches to graph transformation, Part I: Basic concepts and double pushout approach. In Rozenberg, G., ed.: Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations. World Scientific (1997) 163–245
5. Gadducci, F., Montanari, U.: The tile model. In Plotkin, G., Stirling, C., Tofte, M., eds.: Proof, Language and Interaction: Essays in Honour of Robin Milner. MIT Press (1999) To appear. An early version appeared as Tech. Rep. TR-96/27, Dipartimento di Informatica, University of Pisa, 1996. Paper available from `http://www.di.unipi.it/~gadducci/papers/TR-96-27.ps.gz`.
6. Heckel, R.: Open Graph Transformation Systems: A New Approach to the Compositional Modelling of Concurrent and Reactive Systems. PhD thesis, TU Berlin (1998)

On Hierarchical Reconfiguration of Reo Connectors

C. Koehler[1,⋆,⋆⋆], F. Arbab[1], and E.P. de Vink[2]

[1] CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
[2] Technische Universiteit Eindhoven, Den Dolech 2, Eindhoven, The Netherlands

To adapt a software system to new requirements or goals, reconfiguration can be used to restructure the architecture of the system, thereby changing its overall behaviour. In major contrast to conventional approaches that usually involve a complete redeployment, reconfigurations are performed in-place. Algebraic graph transformation [1] describes such structural changes of a system in a concise way and provide a powerful theoretical framework to study reconfigurations.

The coordination paradigm distinguishes between components, which perform the actual computation, and connectors, which coordinate the components [2]. The coordination language Reo [3] uses channels and nodes to construct connectors compositionally. The inherent graph structure of Reo connectors makes graph transformation perfectly suited to model reconfiguration. Fig. 1 shows an example of a reconfiguration rule for a scheduler modeled with Reo. Levels of abstraction can be achieved in Reo via nesting and black-boxing. For example, in Fig. 1, the inner structure of the scheduler and counter are encapsulated, leaving their ports as their interfaces; the three tasks may be considered as atomic components.

We consider to exploit the hierarchical structure of a system's layout to drive the graph transformation. Fig. 1 depicts a system with two active tasks that migrates to a three-task system once the counter reaches a certain value. A lower-level graph transformation adds ports to the scheduler component; a subsequent transformation of the system restores a consistent wiring of the update interface. Thus, driven by the hierarchy, a system reconfiguration decomposes and may trickle down to reconfiguration of its components. Induced topological changes propagate upward, since they require the embedding of the updated subsystem in an actualized connector. Thus, transformation of a Reo network combines the reconfiguration of its components with reconfiguration of the network at a particular level of abstraction. The precise definition of a reconfiguration of a Reo network specifically deals with (i) replacement of hyperedges consisting of the interface nodes of an inner component, (ii) the reconciliation of separate reconfigurations modeled as a vertical composition of transformations.

The proposed reconfiguration of Reo connectors fits in the direct approach of hierarchical graph transformations [4], where the hierarchy is modeled explicitly. In contrast, other approaches [5–7] express the hierarchy indirectly using aggregation edges in an otherwise flat graph. A further characteristic of our approach

⋆ Supported by NWO GLANCE project WoMaLaPaDiA and SYANCO.
⋆⋆ Corresponding author, e-mail christian.koehler@cwi.nl.

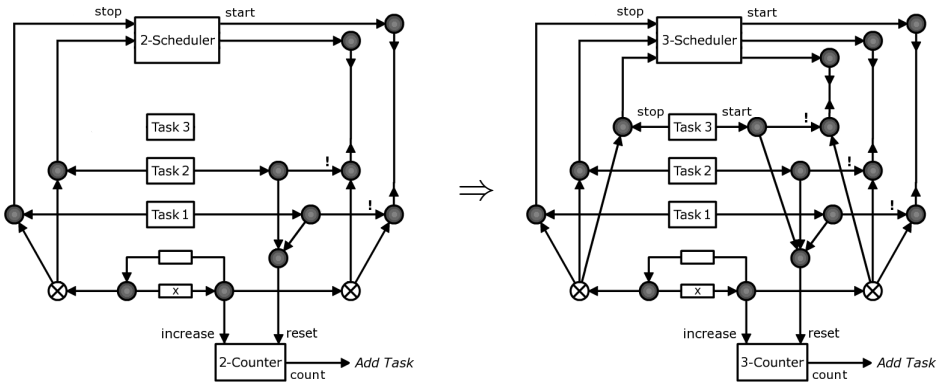


Fig. 1. Reconfiguration rule for a scheduler modeled with Reo.

is that, transformation acts on logical layers of interconnected black-boxes, while composition of transformations combine into higher-level rewrites. In this set-up, disjointness conditions cater for horizontal consistency. By interpreting black-boxed components as hyperedges, a specific notion of composition of transformations arises wherein vertical consistency boils down to interface compatibility. Using the theory of graph transformation, we introduce a hierarchical graph model that, unlike previous approaches, (i) allows a restricted connection of the different hierarchy levels using special interface nodes, (ii) facilitates the application of the double-pushout approach (cf. [1]) to these hierarchical structures, and (iii) supports a hierarchy-driven composition of transformations, separating the reconfigurations of each layer.

Although promising as a first step, further study is necessary to validate our approach of hierarchy-driven reconfiguration of Reo networks.

References

1. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer (2006)
2. Arbab, F.: The IWIM model for coordination of concurrent activities. In: Proc. Coordination'96, LNCS 1061 (1996) 34–56
3. Arbab, F.: Reo: a channel-based coordination model for component composition. Mathematical Structures in Computer Science **14** (2004) 329–366
4. Drewes, F., Hoffmann, B., Plump, D.: Hierarchical graph transformation. Journal of Computer and System Sciences **64** (2002) 249–283
5. Engels, G., Heckel, R.: Graph transformation as a conceptual and formal framework for system modeling and model evolution. In: Proc. ICALP 2000, LNCS 1853 (2000) 127–150
6. Koehler, C., Lewin, H., Taentzer, G.: Ensuring containment constraints in graph-based model transformation approaches. In: Proc. GT-VMT 2007, EASST (2007)
7. Kleppe, A., Rensink, A.: A Graph-Based Semantics for UML Class and Object Diagrams. In: Proc. GT-VMT 2008, EASST (2008)

Tiles for Reo (Extended Abstract) ⋆

Farhad Arbab[1], Roberto Bruni[2], Dave Clarke[1], Ivan Lanese[3], and Ugo Montanari[2]

[1] CWI, Amsterdam, The Netherlands
{farhad,dave}@cwi.nl
[2] Dipartimento di Informatica, Università di Pisa, Italy
{bruni,ugo}@di.unipi.it
[3] Dipartimento di Scienze dell'Informazione, Università di Bologna, Italy
lanese@cs.unibo.it

Reo [1, 2] is an exogenous coordination model for software components. It is based on channel-like connectors that mediate the flow of data and signals among components. Notably, a small set of point-to-point primitive connectors is sufficient to express a large variety of interesting constraints over the behaviour of connected components, including various forms of mutual exclusion, synchronisation, alternation, and context-dependency. In fact, components and primitive connectors can be composed in a circuit fashion via suitable attach points, called Reo nodes. Typical primitive connectors are the synchronous / asynchronous / lossy channel and the asynchronous one-place buffer. The informal Reo's semantics has been matched by several proposals of formalisation, exploiting co-algebraic techniques [3], constraint-automata [4], and colouring tables [5].

Figure 1 shows a small but non-trivial example of Reo circuit for modelling an *exclusive router*, together with the explanation of how the different kinds of connectors are drawn as arrows. If some datum n is written on A then it flows to B on the synchronous channel s_A. Node B must push (copies of) n on the three outgoing channels ls_C, ls_D and sd. The datum can get lost on ls_C or on ls_D, because they are lossy, but not on both. In fact the synchronous drain sd can get n from B only if E can provide another datum. This is possible only if E receives the datum from C (via s_C) or from D (via s_D), and E is not allowed to take the datum from both synchronous channels. Therefore it must be the case that exactly one node between C and D receives n, which is then forwarded either to F or to G. The example suggests that the propagation of constraints can introduce some context-awareness in certain parts of the circuit (see [5]).

We aim to show that the Tile Model [6] offers a flexible and adequate semantic setting for Reo. The name 'tile' is due to the graphical representation of such rules (see Fig. 2). The tile α states that the *initial configuration* s can be triggered by the event a to reach the *final configuration* t, producing the *effect* b. Tiles can be composed in three different ways to generate larger steps: (i) horizontally (synchronisation), when the effect of one tile matches the trigger for another tile; (ii) vertically (composition in time), when the final configuration of one tile matches the initial configuration of another tile; and (iii) in parallel (concurrency).

⋆ Research supported by the project FET-GC II IST-2005-16004 SENSORIA, by the Italian FIRB project TOCAI, by the Dutch NWO project n. 612.000.316 C-Quattro, and by the bilateral German-Dutch DFG-NWO project n. 600.643.000.05N12 SYANCO.

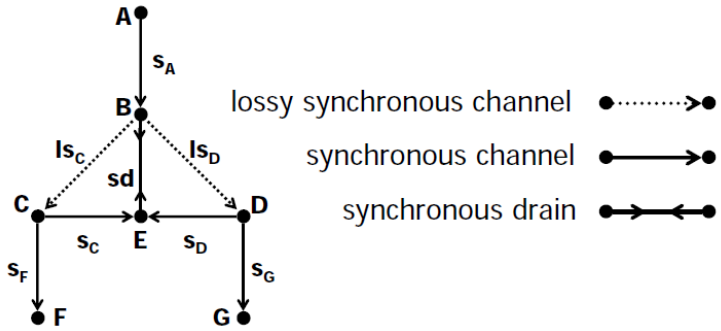


Fig. 1. Exclusive router (from A to either F or G) as a Reo circuit

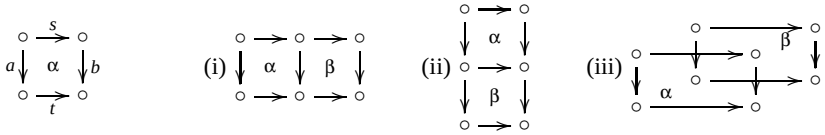


Fig. 2. Examples of tiles and their composition.

Tiles resemble Gordon Plotkin's SOS inference rules [7], but they can be composed horizontally, vertically and in parallel to build larger proof steps. They take inspiration from Andrea Corradini and Ugo Montanari's Structured Transition Systems [8] and generalise Kim Larsen and Liu Xinxin's context systems [9], by allowing for more general rule formats. The tile model also extends José Meseguer's rewriting logic [10] (in the non-conditional case) by taking into account rewrite with side effects and rewrite synchronisation. As rewriting logic, the tile model admits a purely logical formulation, where tiles are seen as sequents subject to certain inference rules.

The definition of a tile semantics for Reo has specific features:

- Concurrency aspects can be taken into account. In fact, tiles have been designed around concurrent systems, hence it is common to consider a monoidal structure of states that gives rise to a monoidal double-category of concurrent computations.
- Tile bisimilarity and tile trace equivalence offer standard abstract equivalences.
- Meta-theoretical results can be exploited to guarantee that tile bisimilarity is a congruence, thus reconciling the algebraic and co-algebraic views of connectors.

The case of stateless connectors has been already considered in [11], in which case a normal form axiomatisation is available for tile bisimilarity, that coincides with tile trace equivalence and with the 2-colouring semantics of [5]. Roughly, Reo nodes and connectors are represented as hyper-edges (with typed incoming and outgoing tentacles) that can be composed sequentially (horizontally) and in parallel by connecting their tentacles. The semantics of each connector c is defined by suitable basic tiles whose initial configuration is the hyper-edge c and whose triggers and effects define how the

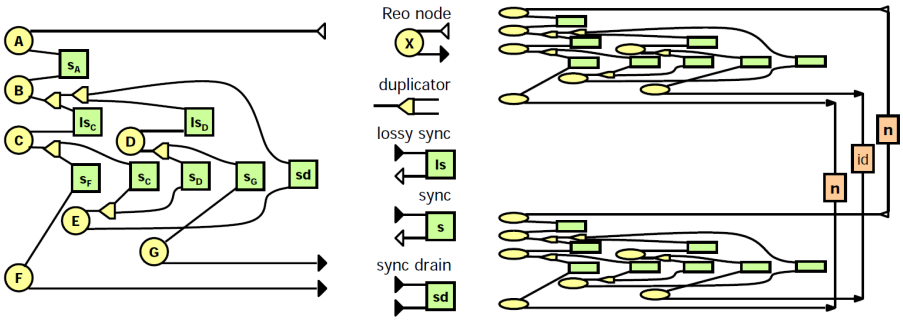


Fig. 3. Tile model for the exclusive router circuit

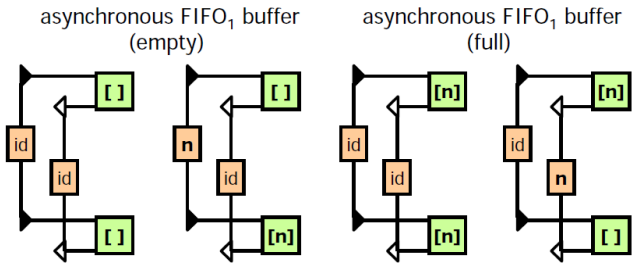


Fig. 4. Basic tiles for asynchronous $FIFO_1$ buffer

data can flow through *c*. Figure 3 shows the configuration diagram that corresponds to the exclusive router, together the explanation of how the different kinds of hyper-edges correspond to Reo elements and with an example of derived tile composition. To improve readability we use different shapes and colors for nodes, channels and vertical observations. A duplicator is a special kind of hyper-edge that allows to attach multiple connectors to the same node. White triangles are used to type incoming attach points and black triangles to type outgoing attach points: they are oriented according to the flow of data. The derived tile composition in Fig. 3 is obtained by horizontal and parallel pasting of basic tiles. The overall trigger is void, because the configuration has no attach point on the left. The overall effect models the routing of a datum *n* from the incoming interface of *A* to the outgoing interface of *F*, with *G* idle.

During the talk we will show that the semantics given in [11] can be extended to take into account stateful connectors, like one-place buffers (see Fig. 4) and, more importantly, it can deal with the finer 3-colouring semantics of [5], where the causes of inhibited interactions can be tracked. In the presence of stateful connectors, one advantage of tiles w.r.t. colouring tables is that the state of the connector after each step is made explicit in the final configurations of basic tiles (while it is not shown in colouring tables).

Finally, we observe that the Tile Model can offer a uniform setting for representing not only the ordinary execution of Reo systems but also dynamic reconfiguration

strategies in the style of [12–14], thus reconciling relevant aspects that were dealt with separately in previous proposals.

References

1. Arbab, F.: Reo: A channel-based coordination model for component composition. Math. Struct. in Comput. Sci. **14**(3) (2004) 1–38
2. CWI: A repository of Reo connectors. http://homepages.cwi.nl/~proenca/webreo/.
3. Arbab, F., Rutten, J.J.M.M.: A coinductive calculus of component connectors. In Wirsing, M., Pattinson, D., Hennicker, R., eds.: Proceedings of WADT 2002. Volume 2755 of Lect. Notes in Comput. Sci., Springer (2002) 34–55
4. Baier, C., Sirjani, M., Arbab, F., Rutten, J.J.M.M.: Modeling component connectors in Reo by constraint automata. Sci. Comput. Program **61**(2) (2006) 75–113
5. Clarke, D., Costa, D., Arbab, F.: Connector colouring I: Synchronisation and context dependency. Sci. Comput. Program **66**(3) (2007) 205–225
6. Gadducci, F., Montanari, U.: The tile model. In Plotkin, G., Stirling, C., Tofte, M., eds.: Proof, Language and Interaction: Essays in Honour of Robin Milner. MIT Press (2000) 133–166
7. Plotkin, G.D.: A structural approach to operational semantics. J. Log. Algebr. Program. **60-61** (2004) 17–139
8. Corradini, A., Montanari, U.: An algebraic semantics for structured transition systems and its application to logic programs. Theoret. Comput. Sci. **103** (1992) 51–106
9. Larsen, K.G., Xinxin, L.: Compositionality through an operational semantics of contexts. In Paterson, M., ed.: Proceedings of ICALP'90. Volume 443 of Lect. Notes in Comput. Sci., Springer (1990) 526–539
10. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoret. Comput. Sci. **96** (1992) 73–155
11. Bruni, R., Lanese, I., Montanari, U.: A basic algebra of stateless connectors. Theoret. Comput. Sci. **366**(1-2) (2006) 98–120
12. Clarke, D.: Reasoning about connector reconfiguration II: Basic reconfiguration logic. In Arbab, F., Sirjani, M., eds.: Proceedings of FSEN'05. Volume 159 of Elect. Notes in Th. Comput. Sci., Elsevier Science (2006) 61–77
13. Koehler, C., Lazovik, A., Arbab, F.: Connector rewriting with high-level replacement systems. In Canal, C., Poizat, P., Viroli, M., eds.: Proceedings of FOCLASA'07. Elect. Notes in Th. Comput. Sci., Elsevier Science (2007)
14. Koehler, C., Costa, D., Proenca, J., Arbab., F.: Reconfiguration of Reo connectors triggered by dataflow. In Ermel, C., Heckel, R., de Lara, J., eds.: Proceedings of GT-VMT'08. Elect. Communic. of the EASST, EASST (2008)

Autonomous Units and Their Semantics – The Concurrent Case*

Hans-Jörg Kreowski and Sabine Kuske

University of Bremen, Department of Computer Science
P.O.Box 33 04 40, 28334 Bremen, Germany
{kreo,kuske}@informatik.uni-bremen.de

In this paper, we introduce and investigate the concurrent semantics of autonomous units. Communities of autonomous units are proposed in [1] as rule-based and graph-transformational devices to model interactive processes that run indepentdently of each other in a common environment. An autonomous unit has a goal that it tries to reach, a set of rules the applications of which provide its actions, and a control condition which regulates the choice of actions to be performed actually. Each autonomous unit decides about its activities on its own right depending on the state of the environment and the possibility of rule applications, but without direct influence of other ongoing processes.

In [2], the sequential semantics of autonomous units is studied. In this case, a single unit can act at a time, while all other units must wait. This yields sequences of rule applications interleaving the activities of the various units. Typical examples of this kind are board games with several players who can perform their moves in turn. In [3], the process steps are given by the application of parallel rules that are composed of the rules of the active units. In this way, units can act simultaneously providing a kind of parallelism which is known from Petri nets, cellular automata, multi-agent systems, and graph transformation.

The sequential and the parallel semantics of communities of autonomous units are based on sequential and parallel derivations resp. Both are composed of derivation steps. In other words, the semantics assumes implicitly the existence of a global clock to cut the run of the whole system into steps. But this is not always a realistic assumption, because the environment may be very large and - more important - the idea of autonomy conflicts with the regulation by a global clock. For example, trucks in a large transport network upload, move, and deliver asynchronously, and do not operate in simultaneous steps and even less in interleaved sequential steps.

The concurrent semantics avoids the assumption of a global clock. The actions of units are no longer totally ordered or simultaneous, but only partially ordered. The partial order reflects causal dependencies meaning that one action takes place before another action if the latter needs something that the former provides. The causal dependency relation of the concurrent semantics

* The authors would like to acknowledge that their research is partially supported by the Collaborative Research Centre 637 (Autonomous Cooperating Logistic Processes: A Paradigm Shift and Its Limitations) funded by the German Research Foundation (DFG).

of autonomous units is compared with shift independency known from graph transformation, and concurrent processes in the present approach are related to canonical derivations (see, e.g., [4]). Moreover, we show that processes of condition/event systems are special cases of the concurrent semantics of autonomous units, and we relate this semantics with other approaches to concurrency (see, e.g., [5–7]).

References

1. Hölscher, K., Klempien-Hinrichs, R., Knirsch, P., Kreowski, H.J., Kuske, S.: Autonomous units: Basic concepts and semantic foundation. In Hülsmann, M., Windt, K., eds.: Understanding Autonomous Cooperation and Control in Logistics – The Impact on Management, Information and Communication and Material Flow, Berlin Heidelberg New York, USA, Springer (2007)
2. Hölscher, K., Kreowski, H.J., Kuske, S.: Autonomous units and their semantics — the sequential case. In: Proc. International Conference of Graph Transformation. Volume 4178 of Lecture Notes in Computer Science. (2006) 245–259
3. Kreowski, H.J., Kuske, S.: Autonomous units and their semantics — the parallel case. In Fiadeiro, J., Schobbens, P., eds.: Recent Trends in Algebraic Development Techniques, 18th International Workshop, WADT 2006. Volume 4408 of Lecture Notes in Computer Science. (2007) 56–73
4. Corradini, A., Ehrig, H., Heckel, R., Löwe, M., Montanari, U., Rossi, F.: Algebraic approaches to graph transformation part I: Basic concepts and double pushout approach. In Rozenberg, G., ed.: Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations. World Scientific, Singapore (1997) 163–245
5. Girault, C., Valk, R.: Petri Nets for Systems Engineering. Springer (2003)
6. Diekert, V., Rozenberg, G., eds.: The Book of Traces. World Scientific, Singapore (1995)
7. Ehrig, H., Kreowski, H.J., Montanari, U., Rozenberg, G., eds.: Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 3: Concurrency, Parallelism, and Distribution. World Scientific, Singapore (1999)

Graph Transformation Modules for the Specification of Reactive Systems

Luciana Foss[1], Leila Ribeiro[1], and Andrea Corradini[2]

[1] Instituto de Informática, Universidade Federal do Rio Grande do Sul,
Porto Alegre, Brazil
{lfoss,leila}@inf.ufrgs.br
[2] Dipartimento di Informatica, Universit di Pisa
Pisa, Italy
andrea@di.unipi.it

Reactive systems are composed by autonomous entities that interact with each other. Each component reacts to signals/messages received as input by sending new signals/messages to other components. The accomplishment of a task of the system involves many interactions among its components.

Since reactive systems are naturally modularized into components, it is advisable to use a specification method that is able to describe this structure in a suitable way. One of the main ingredients of a good module concept is a notion of refinement or implementation: it must be possible to describe the behaviour of a module at an abstract level, and to ensure that the module really executes according to this abstract specification (that is, the body of the module implements or is a refinement of its abstract interface). Since we are dealing with reactive systems, the abstract behaviour should describe interaction patterns that a module or component may engage in.

Several methods for design and analysis of reactive systems propose synchronous languages as specification formalism [1, 2], where the time of reaction to an event (receipt of a signal/message) is null. This characteristic is important to simplify the model, but frequently, it does not correspond to the reality, for example in the case of distributed systems, where the communication between components may take some time. Thus, new approaches were introduced to combine synchronous components and asynchronous communication (for example, [3]).

Graph transformation systems (GTSs) are a suitable formalism to specify complex systems, since graphs are used to describe in a natural way the structure of a system focusing on its components and their interconnections. This formalism give us a simple way to describe concurrency, where the rules of the system can be applied in parallel if they are independent. Moreover, due to the use of rules to specify the behavior of the systems, this specification formalism is particularly well-suited for reactive systems: the left-hand sides of rules describe incoming signals and the right-hand side defines the reaction to these signals.

In [4, 5] a notion of transactions for graph transformation systems was introduced. Since GTSs have an asynchronous semantics, the introduction of this notion allows us to specify the synchronisation of internal activities of a component, modelling them as transactions. Thus, at a more abstract level, we can

consider a transaction as an immediate reaction, where the intermediate steps are considered to take a null time. However, describing a whole interaction as one rule if not yet enough to describe abstractly the behavior of a reactive system: the information about the order in which the input signals are consumed and outputs are generated is lost, and consequently it is not possible to characterize faithfully the interaction that takes place to fulfill the task corresponding to a transaction. To solve this problem, exploring the ideas of [6], a new formalism to specify reactive systems was introduced in [7], namely transactional graph transformation systems with dependency relation, equipping transactions with a relation carrying information about the dependencies between consumed (input) and generated (output) items.

In this paper we elaborate further the theory of transactions equipped with dependencies, investigating their relation to transactions without dependencies and their use to specify abstractly the behavior of a system, giving raise to a module concept for reactive systems. A dT-GTS (dependencyTransactional Graph Transformation System) module is defined by an interface and a body that implements it. The interface describes at an abstract level the operations provided by the module, allowing to hide resources that are temporary in the module body, realising abstraction from internal details and giving an independence from particular implementation of operations. Moreover, it describes the interaction of the system with its environment in order to realise their operations: which signals are sent to the environment in reaction to received ones.

References

1. Berry, G.: The foundations of Esterel. In: Proof, language, and interaction: essays in honour of Robin Milner. MIT Press (2000) 425–454
2. Halbwachs, N.: Synchronous programming of reactive systems: a tutorial and commented bibliography. In: International Conference on Computer-aided Verification, CAV '98. Volume 1427 of LNCS., Springer (1998) 1–16
3. Riesco, M., Tuya, J.: Synchronous estelle: just another synchronous language? Electronic Notes in Theoretical Computer Science **88** (2003) 71–86
4. Baldan, P., Corradini, A., Dotti, F., Foss, L., Gadducci, F., Ribeiro, L.: Towards a notion of transaction in graph rewriting. Electronic Notes in Theoretical Computer Science (2008) 1–12
5. Baldan, P., Corradini, A., Foss, L., Gadducci, F.: Graph transactions as processes. In: ICGT 2006 - Graph Transformations. Volume 4178 of LNCS., Springer (2006) 199–214
6. Ribeiro, L., Dotti, F., Santos, O., Pasini, F.: Verifying object-based graph grammars: An assume-guarantee approach. Software and Systems Modeling **5** (2006) 289–312
7. Foss, L., Machado, R., Ribeiro, L.: Graph productions with dependencies. In: SBMF - Simpósio Brasileiro de Métodos Formais. (2007) 128–143

Modeling Data-Dependent Workflows in Mobile Ad-hoc Networks using High-Level Nets and Rules as Tokens

Julia Padberg, Kathrin Hoffmann*, Hartmut Ehrig

Institut für Softwaretechnik und Theoretische Informatik
Technische Universität Berlin, Germany

Modeling workflows is one of the main approaches for adequate software support in business applications. We propose using workflow modeling to ensure satisfactory team cooperation based on mobile ad-hoc networks (MANETs). From a more abstract point of view MANETs consist of mobile nodes which communicate with each other independently from a stable infrastructure, while the topology of the network constantly changes depending on the current position of the nodes and their availability. We present a modeling technique that both enables the modeling of flexible workflows in MANETs and supports changes of the network topology and the subsequent transformation of workflows.

In [1] modeling of workflows in MANETs using algebraic higher-order (AHO) nets has been introduced. AHO nets are Petri nets with complex tokens, namely place/transition nets and a set of rules (see [2]). In more detail, AHO nets are a specific class of algebraic high-level (AHL) nets that combine algebraic specifications (in the sense of [3]) with Petri nets and allow modeling data flow or data changes within the net.

In [4] we have demonstrated the flexibility of AHO-nets concerning different kinds of objects, i.e. we have given a signature and a corresponding algebra modelling the token game of Petri nets and rule-based transformations. For this purpose we use the framework of net transformations [5] that is inspired by graph transformation systems [6]. The basic idea is the stepwise modification of Petri nets by given rules. The rules present a rewriting of nets where the left-hand side is replaced by the right-hand side. As a result not only the follower marking of nets as tokens can be computed but also their structure can be changed by rule application to obtain a new net that taking into account new requirements of the environment. Moreover these activities can be interleaved [2].

Up to now we have only considered low-level Petri nets as tokens in AHO nets. But our experience with the case studies in [7] and [8] has clearly shown the need to integrate data and communication at the level of workflows. The basis of our case study in [8] is the part of the "Johns Hopkins Safety Manual" [9, 10] for responsibilities of a fire on an inpatient unit in the Johns Hopkins Hospital. In more detail each employee gets its own open AHL net and specific rules are provided to modify the sequences of tasks, if necessary. Moreover, the exchange of information is described by input and output places. Depending on

* This work has been partly funded by the research project forMA$_1$NET (see http://tfs.cs.tu-berlin.de/formalnet/) of the German Research Council.

the kind of communication - internal or external - the sending and receiving of messages, respectively, are simulated either by application of communication rules or creation and deletion of tokens.

For the development and analysis of Petri nets we have shown in [11] that AHL nets which additionally include markings form a weak adhesive high-level replacement (HLR) category. Additionally, in [8] open AHL nets and in [12] AHL nets with rules having negative application conditions were shown to be weak adhesive HLR categories. Adhesive HLR categories [6] have been established as a suitable categorical framework for double pushout transformations.

So, we now present an approach that includes the treatment of data and uses AHO nets with data dependent tokens consisting of AHL nets as well as rules and net transformations for changing these AHL nets. Thus the main focus of this contribution is the integration of open AHL nets and corresponding rules with possible negative application conditions as token objects into AHO nets based on a suitable data type part.

This is achieved by providing a suitable signature and algebra for representing AHL nets and their operational behavior as well as rules and their application. Based on vocabularies for transitions and places the signature HLNR-System-SIG involves sorts for the various types of nets, morphisms and rules. The operations concern mainly enabling and firing as well as applicability and transformation. The algebra is based on the underlying algebra of the AHL nets to be represented. The constructions are applied in a small case study based on [8].

References

1. Bottoni, P., De Rosa, F., Hoffmann, K., Mecella, M.: Applying Algebraic Approaches for Modeling Workflows and their Transformations in Mobile Networks. Journal of Mobile Information Systems **2**(1) (2006) 51–76
2. Ehrig, H., Hoffmann, K., Padberg, J., Prange, U., Ermel, C.: Independence of Net Transformations and Token Firing in Reconfigurable Place/Transition Systems. In: Proc. Application and Theory of Petri Nets (ATPN). Volume 4546 of LNCS., Springer (2007) 104–123
3. Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specification 1: Equations and Initial Semantics. Volume 6 of EATCS Monographs on Theoretical Computer Science. Springer, Berlin (1985)
4. Hoffmann, K., Ehrig, H., Mossakowski, T.: High-Level Nets with Nets and Rules as Tokens. In: Proc. Application and Theory of Petri Nets (ATPN). Volume 3536 of LNCS., Springer (2005) 268–288
5. Ehrig, H., Padberg, J.: Graph Grammars and Petri Net Transformations. In: Lectures on Concurrency and Petri Nets Special Issue Advanced Course PNT. Volume 3098 of LNCS., Springer (2004) 496–536
6. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS Monographs in Theoretical Computer Science. Springer (2006)
7. Hoffmann, K., Ehrig, H., Padberg, J.: Flexible modeling of emergency scenarios using reconfigurable systems. In: Proc. of the 10th World Conference on Integrated Design & Process Technology. (2007)

8. Ullrich, C.: Reconfigurable Open AHL Systems. Master's thesis, TU Berlin, to appear as Technical Report (2008)
9. Johns Hopkins Safety Manual: Fire Incident Responsibilities in JHH. (2007)
10. Johns Hopkins Safety Manual: Incipient Fire Response Team for JHH. (2007)
11. Prange, U.: Algebraic High-Level Nets as Weak Adhesive HLR Categories. Electronic Communications of the EASST **2** (2007) 1–13
12. Rein, A.: Reconfigurable Petri Systems with Negative Application Conditions. Master's thesis, TU Berlin, to appear as Technical Report (2008)

A Rewriting Logic Approach to Type Inference

Chucky Ellison, Traian Florin Şerbănuţă and Grigore Roşu

Department of Computer Science, University of Illinois at Urbana-Champaign
{celliso2, tserban2, grosu}@cs.uiuc.edu

Rewriting logic semantics (RLS) was proposed as a programing language definitional framework that unifies operational and algebraic denotational semantics; see [1, 2] and the references there. Once a language is defined as an RLS theory, many generic tools are immediately available for use with no additional cost to the designer. These include a formal inductive theorem proving environment, an efficient interpreter, a state space explorer, and even a model checker. RLS has already been used to define a series of didactic and real languages [1].

In its SOS'05 precursor, [1] proposed using the same rewriting logic technique to define type systems and policy checkers for languages; more precisely, to rewrite integer values to their types and to maintain and incrementally rewrite a program until it becomes a type or other desired abstract value. That idea was further explored by in [2], but not yet used to define complex, polymorphic type systems; it also provides no implementation, no proofs, and no empirical evaluation of the idea. A similar idea has been recently proposed by [3] in the context of Felleisen et al.'s reduction semantics with evaluation contexts [4, 5] and Matthews et al.'s PLT Redex system [6].

In this paper we show how the same rewriting logic semantics (RLS) framework and definitional style employed in giving formal semantics to languages can also be used to define type systems as rewrite logic theories. This way, both the language and its type system(s) can be defined using the same formalism, facilitating reasoning about programs, languages, and type systems.

We use Milner's polymorphic type inferencer $\mathcal{W}$ [7] for the Exp language to exemplify our technique. We give one rewrite logic theory for $\mathcal{W}$ and use it both for proving its correctness against a rewrite theory defining Exp and for obtaining an efficient, executable type-inferencer.

Our definitional style gains modularity by specifying the minimum amount of information needed for a transition to occur, and compositionality by using *strictness* attributes associated to the language constructs, which specify that the semantics of that construct involves certain specified arguments to be evaluated (and their side effects propagated) prior to giving semantics to the construct itself. These allow us, for example, to have the rule for function application corresponding to the one in $\mathcal{W}$ look as follows (assuming the application was declared strict in both arguments):

$$\langle\!\langle \underline{t_1\ t_2} \rangle\!\rangle_k \ \langle\!\langle \frac{\cdot}{t_1 \equiv t_2 \to tvar} \rangle\!\rangle_{eqns} \quad \text{where } tvar \text{ is a fresh type variable}$$

which reads as follows: once all constraints for both sides of an application construct were gathered, the application of t_1 to t_2 will have a new type, *tvar*, with the additional constraint that t_1 is the function type $t_2 \to tvar$.

This work makes three novel contributions:

1. It shows how non-trivial type systems are defined as RLS theories in a uniform way, following the same style used for defining programming languages and other formal analyses for them;
2. It proposes a type soundness proof technique for languages and type systems defined as RLS theories; and
3. It shows that RLS definitions of type systems, when executed on existing rewrite engines, yield competitive type inferencers.

To show that the proposed rewriting approach and the resulting type inferencers scale, Milner's simple language is extended with multiple-binding let and letrec, with lists, and with references and side effects. The resulting type inferencer, able to detect weak polymorphism, is only slightly slower than the one for Milner's simpler language.

All these show that rewriting logic is amenable for defining feasible type inferencers for programming languages and proving type soundness for those definitions. Doing the proof of soundness for $\mathcal{W}$ and other systems have led us to believe that this kind of proofs should be easily mechanizable. We strongly adhere to the program proposed by the PoplMark Challenge [8], and would like to approach it using the proposed novel methodology.

Below we include the K definition of the $\mathcal{W}$ type inferencer, including the syntax of Exp for $\mathcal{W}$, the configuration, unification rules, and complete typing rules. Information about the K syntax can be found in [2].

$$
\begin{array}{rcll}
\mathit{Var} & ::= & \text{standard identifiers} & \\
\mathit{Exp} & ::= & \mathit{Var} \mid \ldots \text{ add basic values (Bools, ints, etc.)} & \\
& \mid & \lambda\, \mathit{Var}\,.\, \mathit{Exp} & \\
& \mid & \mathit{Exp}\ \mathit{Exp} & [\mathit{strict}] \\
& \mid & \mu\, \mathit{Var}\,.\, \mathit{Exp} & \\
& \mid & \mathsf{if}\ \mathit{Exp}\ \mathsf{then}\ \mathit{Exp}\ \mathsf{else}\ \mathit{Exp} & [\mathit{strict}] \\
& \mid & \mathsf{let}\ \mathit{Var} = \mathit{Exp}\ \mathsf{in}\ \mathit{Exp} & [\mathit{strict}(2)] \\
& \mid & \mathsf{letrec}\ \mathit{Var}\ \mathit{Var} = \mathit{Exp}\ \mathsf{in}\ \mathit{Exp} & [\mathsf{letrec}\ f\ x = e\ \mathsf{in}\ e' = \mathsf{let}\ f = \mu f.(\lambda x.e)\ \mathsf{in}\ e'] \\
\\
\mathit{K} & ::= & \cdots \mid \mathit{Type} \to \mathit{K} & [\mathit{strict}(2)] \\
\mathit{Result} & ::= & \mathit{Type} & \\
\mathit{TEnv} & ::= & \mathit{Map}[\mathit{Name}, \mathit{Type}] & \\
\mathit{Type} & ::= & \ldots \mid \mathit{let}(\mathit{Type}) & \\
\mathit{ConfigItem} & ::= & (\!|\mathit{K}|\!)_{k} \mid (\!|\mathit{TEnv}|\!)_{tenv} \mid (\!|\mathit{Eqns}|\!)_{eqns} \mid (\!|\mathit{TypeVar}|\!)_{nextType} & \\
\mathit{Config} & ::= & \mathit{Type} \mid [\![\mathit{K}]\!] \mid [\![\mathit{Set}[\mathit{ConfigItem}]]\!] & \\
\mathit{Type} & ::= & \ldots \mid \mathit{int} \mid \mathit{bool} \mid \mathit{Type} \mapsto \mathit{Type} \mid \mathit{TypeVar} & \\
\mathit{Eqn} & ::= & \mathit{Type} \equiv \mathit{Type} & \\
\mathit{Eqns} & ::= & \mathit{Set}[\mathit{Eqn}] &
\end{array}
$$

$$
\begin{array}{l}
(t \equiv t) \to \cdot \\
(t_1 \mapsto t_2 \equiv t_1' \mapsto t_2') \to (t_1 \equiv t_1'),\ (t_2 \equiv t_2') \\
(t \equiv t_v) \to (t_v \equiv t) \quad \text{when } t \notin \mathit{TypeVar} \\
t_v \equiv t,\ t_v \equiv t' \to t_v \equiv t,\ t \equiv t' \quad \text{when } t, t' \neq t_v
\end{array}
$$

$t_v \equiv t,\ t'_v \equiv t' \to t_v \equiv t,\ t'_v \equiv t'[t_v \leftarrow t]$
when $t_v \neq t'_v,\ t_v \neq t,\ t'_v \neq t',$ and $t_v \in vars(t')$

$[\![e]\!] = [\![(\!|e|\!)_k\ (\!|\cdot|\!)_{tenv}\ (\!|\cdot|\!)_{eqns}\ (\!|t_0|\!)_{nextType}]\!]$
$[\![(\!|(\!|t|\!)_k\ (\!|\gamma|\!)_{eqns}|\!)]\!] = \gamma[t]$
$i \to int,\ true \to bool,\ false \to bool,$ (and similarly for all the other basic values)
$(\!|\underline{t_1 + t_2}|\!)_k\ (\!|\underline{\quad\cdot\quad}_{t_1 \equiv int,\ t_2 \equiv int}|\!)_{eqns}$ (and similarly for all the other standard operators)
$(\!|\underline{\quad x \quad}_{(\gamma[t])[tl \leftarrow tl']}|\!)_k\ (\!|\eta|\!)_{tenv}\ (\!|\gamma|\!)_{eqns}\ (\!|\underline{\quad t_v \quad}_{t_v + |tl|}|\!)_{nextType}$
when $\eta[x] = let(t),\ tl = vars(\gamma[t]) - vars(\eta)$
and $tl' = t_v \ldots (t_v + |tl| - 1)$ $(\!|\underline{\ x\ }_{\eta[x]}|\!)_k\ (\!|\eta|\!)_{tenv}$ when $\eta[x] \neq let(t)$
$(\!|\underline{\quad \lambda x.e \quad}_{(t_v \to e) \curvearrowright restore(\eta)}|\!)_k\ (\!|\underline{\ \eta\ }_{\eta[x \leftarrow t_v]}|\!)_{tenv}\ (\!|\underline{\ t_v\ }_{t_v + 1}|\!)_{nextType}$
$(\!|\underline{t_1\ t_2}|\!)_k\ (\!|\underline{\quad\cdot\quad}_{t_1 \equiv t_2 \to t_v}|\!)_{eqns}\ (\!|\underline{\ t_v\ }_{t_v + 1}|\!)_{nextType}$
$(\!|\underline{\quad \mu x.e \quad}_{e \curvearrowright ?_=(t_v) \curvearrowright restore(\eta)}|\!)_k\ (\!|\underline{\ \eta\ }_{\eta[x \leftarrow t_v]}|\!)_{tenv}\ (\!|\underline{\ t_v\ }_{t_v + 1}|\!)_{nextType}$
$(\!|t \to \underline{?_= t_v}_{\cdot}|\!)_k\ (\!|\underline{\ \cdot\ }_{t_v \equiv t}|\!)_{eqns}$
$(\!|\underline{\ \mathsf{let}\ x = t\ \mathsf{in}\ e\ }_{e \curvearrowright restore(\eta)}|\!)_k\ (\!|\underline{\quad \eta \quad}_{\eta[x \leftarrow let(t)]}|\!)_{env}$
$(\!|\underline{\mathsf{if}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2}_{t_1}|\!)_k\ (\!|\underline{\quad\cdot\quad}_{t \equiv bool,\ t_1 \equiv t_2}|\!)_{eqns}$

References

1. Meseguer, J., Rosu, G.: The rewriting logic semantics project. J. TCS **373**(3) (2007) 213–237
2. Rosu, G.: K: A rewrite-based framework for modular language design, semantics, analysis and implementation. Technical Report UIUCDCS-R-2006-2802, Computer Science Department, University of Illinois at Urbana-Champaign (2006)
3. Kuan, G., MacQueen, D., Findler, R.B.: A rewriting semantics for type inference. In: ESOP '07. Volume 4421 of LNCS., Springer (2007) 426–440
4. Felleisen, M., Hieb, R.: A revised report on the syntactic theories of sequential control and state. J. TCS **103**(2) (1992) 235–271
5. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Information and Computation **115**(1) (1994) 38–94
6. Matthews, J., Findler, R.B., Flatt, M., Felleisen, M.: A visual environment for developing context-sensitive term rewriting systems. In: RTA '04. Volume 3091 of LNCS., Springer (2004) 301–311
7. Milner, R.: A theory of type polymorphism in programming. J. Computer and System Sciences **17**(3) (1978) 348–375
8. Aydemir, B.E., Bohannon, A., Fairbairn, M., Foster, J.N., Pierce, B.C., Sewell, P., Vytiniotis, D., Washburn, G., Weirich, S., Zdancewic, S.: Mechanized metatheory for the masses: The PoplMark challenge. In: TPHOLs '05. Volume 3603 of LNCS., Springer (2005) 50–65

Term Logic

Andrei Popescu and Grigore Roşu

University of Illinois at Urbana-Champaign
{popescu2,grosu}@uiuc.edu

We introduce Term Logic (TL), a logic for specifying λ-calculi. TL has the following features:

- It is a natural generalization of first-order logic allowing terms with bindings.
- It has a notion of model that generalizes the Henkin models of λ-calculus and validates its syntactic deduction rules.
- It accommodates direct *adequate* specifications of λ-calculi as particular theories, in a similar way in which group theory is a particular theory of first-order equational logic; thus TL relates to its specified calculi more as a *generalization* than as an *encoding*.
- It allows, via a schematic extension of its deductive system with quantification over terms, for *meta-reasoning* about the specified calculi by adding suitable extra axioms to the specification; notably, a uniform induction principle, validated by all reachable models, can be stated in our logic.

Sample specification in TL - simply-typed λ-calculus. Two sorts: *data* and *type*. Operations: $b : \mathit{type}$, $\mathit{Arrow} : \mathit{type} \to \mathit{type}$ non-binding, $\mathit{Lam} : \mathit{data} \times \mathit{type} \to \mathit{data}$ binding on the first argument. Relations: $\mathit{typeOf} : \mathit{data} \to \mathit{type}$, $\mathit{reduce} : \mathit{data} \to \mathit{data}$. Concrete syntax: $P \to P'$ for $\mathit{Arrow}(P, P')$, $Q\,Q'$ for $\mathit{App}(Q, Q')$ $\lambda x : P.Q$ for $\mathit{Lam}(x.Q, P)$, $Q : P$ for $\mathit{typeOf}(Q, P)$, $Q \rightsquigarrow Q'$ for $\mathit{reduce}(Q, Q')$. Axioms (where t, s are type variables, x, y, z, x', y' are data variables, and X, Y, Z are data term-variables, and $\Rightarrow$ is the TL implication):

TpAbs: $\forall t, s, X.\ (\forall x.\ x : t \Rightarrow X : s) \ \Rightarrow\ (\lambda x{:}t.X) : t \to s$
TpApp: $\forall x, y, t, s.\ x : t \to s \ \wedge\ y : t \ \Rightarrow\ x\,y : s$
RedBeta: $\forall x, X.\ (\lambda x{:}t.X)x \ \rightsquigarrow\ X$
RedAppL: $\forall x, y, x'.\ x \rightsquigarrow x' \ \Rightarrow\ x\,y \rightsquigarrow x'\,y$
RedAppR: $\forall x, y, y'.\ y \rightsquigarrow y' \ \Rightarrow\ x\,y \rightsquigarrow x\,y'$
RedXi: $\forall X, Y, t.\ (\forall z.\ X \rightsquigarrow Y) \ \Rightarrow\ \lambda z{:}t.X \rightsquigarrow \lambda z{:}t.Y$

In (the schematic extension of) TL, the semantics of term variables (capitalized) are mappings from environments to values in certain domains, while the semantics of regular variables (lower cases) are mere values. Syntactically, free regular variables may be *substituted* (in a capture-avoiding fashion) by terms, while term variables may be simply *replaced* (possibly causing captures) by terms. Notice that bindings occur at both the term and formula level.

The above axioms capture adequately the operational aspects of simply-typed λ-calculus, w.r.t. both typing and reduction. Namely:

- for any typing context $x_1 : P_1, \ldots, x_n : P_n$, type term P and data term Q, $x_1 : P_1, \ldots, x_n : P_n \vdash Q : P$ iff $\forall x_1, \ldots, x_n.\ x_1 : P_1 \wedge \ldots \wedge x_n : P_n \Rightarrow Q : P$ is deducible in TL from the stated axioms
- for any data terms Q, Q', Q is reducible to Q' in the original calculus iff $\forall x_1, \ldots, x_n.\ Q \rightsquigarrow Q'$, where $x_1, \ldots, x_n$ are the free variables of Q, Q', is deducible in TL from the stated axioms.

To make an analogy with classical logic specification, in the same way the equations $\forall x.\ x+0 = x$ and $\forall x, y.\ x+(suc\ y) = suc(x+y)$ capture the "operational" aspects of natural numbers with successor and addition, in that for any two ground terms P_1, P_2 involving these operations, $P_1 = P_2$ holds for the set of natural numbers iff it is deducible in first-order logic from the mentioned equations. If one wants the ability to establish further properties of the natural numbers in first-order logic, one needs additional axioms, such as the injectivity of successor and an induction scheme. Likewise, we can extend our specification to capture additional properties about the system, such as structural induction and reversed reduction rules: (Notice this dual view that allows a fruitful analogy with classical first-order specifications: the facts expressible *in* the calculus are a particular form of statements *about* the calculus.)

DataInd: $(\forall\, x, y.\ \ \varphi(x) \wedge \varphi(y) \Rightarrow \varphi(x\, y)) \wedge (\forall\, t, X.\ (\forall x.\varphi(X)) \Rightarrow \varphi(\lambda x\!:\!t.\ X)) \Rightarrow \forall x.\ \varphi(x)$

RedAppRev:
$\forall x, y, z.\ \ x\, y \rightsquigarrow z\ \Rightarrow\ (\exists x'.\ z = x'\, y\ \wedge\ x \rightsquigarrow x') \vee (\exists y'.\ z = x\, y'\ \wedge\ y \rightsquigarrow y') \vee (\exists Z(\_ : data), t.\ z = Z(y)\ \wedge\ x = \lambda y\!:\!t.Z(y))$

XiRev: $\forall\, X, Y, t.\ \ \lambda z\!:\!t.X \rightsquigarrow \lambda z\!:\!t.Y\ \Rightarrow\ (\forall\, z.\ X \rightsquigarrow Y)$

In the sentence RedAppRev above, $Z(\_ : data)$ is a "pointed" term variable, i.e., it represents a term together with an emphasized data variable (which may or may not occur free in that term). Whenever a term variable is quantified in a formula as "pointed" by an arity (i.e., sequence of sorts), its later use in the scope of this quantifier should always indicate terms for substituting the assumed variables. Interestingly, most of the times we can do without pointed terms, in situations that traditionally require some form of instantiation/substitution. For example, the axiom RedBeta above avoids pointed terms, but could have been expressed equivalently in a manner more faithful to its original form from the represented calculus, as $\forall\, y, X(\_ : data).\ \ (\lambda x\!:\!t.X(x))\, y\ \rightsquigarrow\ X(y)$.

Assuming the above, and other obvious "rule-reversing" axioms similar to RedAppRev and XiRev (henceforth referred to as "rev-axioms"), we can prove in TL properties like subject reduction:

SubjRed: $\forall\, x, y, t.\ \ x : t\ \wedge\ x \rightsquigarrow y\ \Rightarrow\ y\!:\!t$

The (informal version of the) proof goes as follows. We prove $\forall x.\, \varphi(x)$ using DataInd, where $\varphi(x)$ is $\forall\, y, t.\ \ x : t\ \wedge\ x \rightsquigarrow y\ \Rightarrow\ y\!:\!t$.

- Fix x and y. Assume $\varphi(x)$ and $\varphi(y)$. We need to show $\varphi(x\,y)$. For this, fix z and t and assume $((x\,y) : t$ and $x\,y \rightsquigarrow z$. By the rev-axioms, we obtain t_1 where (Fact1) $x : t_1 \to t$ and (Fact2) $y : t_1$. We have the following cases, according to RedAppRev:
  1. $x \rightsquigarrow x'$ and $z = x'\,y$ for some x'. Then $x' : t_1 \to t$ by the induction hypothesis and Fact1, and, with Fact2 and TpApp, we obtain $z = (x'\,y) : t$, as desired.
  2. $y \rightsquigarrow y'$ and $z = x\,y'$ for some y'. Then $y' : t_1$ by the induction hypothesis and Fact2, and, with Fact1 and TpApp, we obtain $z = x\,y' : t$, as desired.
  3. $z = Z(y)$ and $x = \lambda u : t.\, Z(u)$ for some $Z(\_)$. From Fact1 and the rev-axioms we obtain $\forall u.\, u : t_1 \Rightarrow Z(u) : t$. In particular, with Fact2, we obtain $z = Z(y) : t$, as desired.
- Fix t_1 and X. Assume $\forall x.\, \varphi(X)$. We need $\varphi(\lambda x{:}t_1.X)$. For this, fix z, t and assume $(\lambda x{:}t_1.\, X) \rightsquigarrow z$ and $(\lambda x{:}t_1.X) : t_1$. By the rev-axioms, $t = t_1 \to t_2$ and $\forall x.\, x : t_1 \Rightarrow X : t_2$ for some t_2. Moreover, by Xi-Rev, there exists a $Z(\_)$ such that $z = \lambda x\,.Z(x)$ and $\forall x.X \rightsquigarrow Z(x)$. With the induction hypothesis, we obtain $\forall x.\, Z(x) : t_2$, hence, via TpApp, $z : t_1 \to t_2$, as desired.

Related work. Notable frameworks for representing λ-calculi are Higher-Order Abstract Syntax (HOAS) (in its various forms) [1] [2] and Nominal Logic [3]. Our proposed specification style in TL retains most of the advantages of HOAS (object-level bindings as meta-level bindings, the absence of object-level names as first-class citizens) while at the same time allowing for meta-reasoning, known to be problematic in (strong) HOAS; the latter feature brings TL closer to more first-order approaches like weak HOAS and nominal logic. Our TL models have similarities with the structures built of *explicitly closed families* and *functionals* of [4], the *binding algebras* of [5] and the *substitution algebras* of [6]. [7] presents, in the context of functor categories, an induction principle similar to the one proposed here.

References

1. Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. In: Proc. 2nd LICS Conf., IEEE, Computer Society Press (1987) 194–204
2. Pfenning, F., Elliot, C.: Higher-order abstract syntax. In: PLDI '88, ACM Press (1988) 199–208
3. Pitts, A.M.: Nominal logic: A first order theory of names and binding. In: TACS'01. Volume 2215 of Lecture Notes in Computer Science. (2001) 219–242
4. Aczel, P.: Frege structures and notations in propositions, truth and set. In: The Kleene Symposium, North Holland (1980) 31–59
5. Sun, Y.: An algebraic generalization of Frege structures - binding algebras. Theor. Comput. Sci. **211**(1-2) (1999) 189–232
6. Fiore, M., Plotkin, G., Turi, D.: Abstract syntax and variable binding (extended abstract). In: Proc. 14th LICS Conf., IEEE, Computer Society Press (1999) 193–202
7. Hofmann, M.: Semantical analysis of higher-order abstract syntax. In: LICS '99: Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science. (1999) 204

Rewriting diagrams for computing and interpreting classical logic

Pierre Lescanne and Dragiša Žunić

Université de Lyon, ENS de Lyon, CNRS (LIP), 46 allée d'Italie, 69364 Lyon, France

We present two calculi which computationally interpret sequent calculus style classical logic. The first interpretation is a term-calculus from which we derive a more abstract diagrammatic model.

The first calculus, called ${}^*\mathcal{X}$, captures the structure of sequent proofs. It has been designed to provide a Curry-Howard correspondence with Gentzens sequent system G_1 for classical logic, which is characterized by explicit structural rules *weakening* and *contraction*. The design of the calculus reveals the role of the structural rules in the process of proof-transformation, i.e., on the computational side they appear as terms called *eraser* and *duplicator*, respectively, as they are used to implement erasure and duplication. The computation in ${}^*\mathcal{X}$ corresponds to cut-elimination in the sequent calculus. ${}^*\mathcal{X}$-terms are linear.

This calculus is obtained by applying the ideas from the intuitionistic field, namely the design of the λlxr-calculus [1], which is an extension of the λx-calculus with an eraser and a duplicator, into the $\mathcal{X}$ calculus [2, 3] and Urban's calculus [4, 5]. To the best of our knowledge, it is the first calculus for classical logic which explicitly implements erasure and duplication. The existing classical calculi are not entirely explicit about these operations, although they are important in both theory and implementation. The ${}^*\mathcal{X}$-syntax carries a high level of details, which enables us to naturally derive a more abstract model, that is to derive the syntax and the rules of a diagrammatic calculus.

The second calculus, called ${}^d\mathcal{X}$, is a *diagrammatic calculus* for classical logic. In some sense this is the natural continuation of the previous part; thanks to linearity and the presence of erasers and duplicators, ${}^*\mathcal{X}$-terms can be seen as two dimensional diagrams. This means that we define a diagrammatic syntax and the reduction relation is given by the rewriting rules over those diagrams.

The static aspects of this calculus have recently been defined by Robinson [6] as proof-nets for classical logic, but the dynamic aspects (computation) have not yet been presented. The diagrams have been inspired by Girard's proof-nets for linear logic, although there is a difference in a sense that we search for a system which is in direct relation with standard sequent system for classical logic. ${}^d\mathcal{X}$ abstracts away from unessential details (syntactic bureaucracy) which are unavoidable in a formalism like a sequent calculus. If we have in mind the importance of visual images to human cognitive activities, it is not difficult to motivate the study of ${}^d\mathcal{X}$.

The calculi ${}^*\mathcal{X}$ and ${}^d\mathcal{X}$ are not in one-to-one correspondence since more terms corresponds to one diagram. A detailed study of the relation between the two calculi can be found in [7].

Both calculi are non-deterministic and non-confluent. The reduction rules satisfy *free names* preservation (interface preservation), *type* preservation (computation can be seen as proof-transformation), and the preservation of *linearity* of terms.

The long version of this work is available on the web [8].

References

1. Kesner, D., Lengrand, S.: Ressource operators for lambda-calculus. Information and Computation **205**(4) (2007) 419–473
2. van Bakel, S., Lengrand, S., Lescanne, P.: The language $\mathcal{X}$: circuits, computations and classical logic. In: Proc.9th Italian Conf. on Theoretical Computer Science (ICTCS'05). Volume 3701 of Lecture Notes in Computer Science. (2005) 81–96
3. van Bakel, S., Lescanne, P.: Computation with classical sequents. Mathematical Structures in Computer Science (2007) To appear.
4. Urban, C.: Classical Logic and Computation. PhD thesis, University of Cambridge (2000)
5. Urban, C., Bierman, G.M.: Strong normalisation of cut-elimination in classical logic. In: Typed Lambda Calculus and Applications. Volume 1581 of Lecture Notes in Computer Science., Springer-Verlag (1999) 365–380
6. Robinson, E.: Proof nets for classical logic. Journal of Logic and Computation **13**(5) (2003) 777–797
7. Žunić, D.: Computing with Sequents and Diagrams in Classical Logic-Calculi ${}^{*}\mathcal{X}$, ${}^{d}\mathcal{X}$ and ${}^{©}\mathcal{X}$. PhD thesis, ENS de Lyon (December 2007)
8. Lescanne, P., Žunić, D.: Rewriting diagrams for computing and interpreting classical logic. http://perso.ens-lyon.fr/dragisa.zunic/wadt08.pdf (2008)

Translating Dependently-Typed Logic to First-Order Logic

Kristina Sojakova and Florian Rabe

Jacobs University Bremen

Abstract. DFOL is a logic that extends first-order logic with dependent types. We give a translation from DFOL to FOL formalized as an institution comorphism and show that it admits the model expansion property. This property together with the borrowing theorem implies the soundness of borrowing — a result that enables us to reason about entailment in DFOL by using automated tools for FOL. In addition, the translation permits us to deduce properties of DFOL such as completeness, compactness, and existence of free models from the corresponding properties of FOL, and to regard DFOL as a fragment of FOL. Future work will focus on the integration of the translation into the specification and translation tool Hets.

Dependent type theory, DTT, ([1]) provides a very elegant language for many applications ([2, 3]). However, its definition is much more involved than that of simple type theory because all well-formed terms, types, and their equalities must be defined in a single joint induction. Several quite complex model classes, mainly related to locally cartesian closed categories, have been studied to provide a model theory for DTT (see [4] for an overview).

Many of the complications disappear if dependently-typed extensions of first-order logic are considered, i.e., systems that have dependent types, but no (simple or dependent) function types. Such systems were investigated in [5], [6], and [7]. They provide very elegant axiomatizations of many important mathematical theories such as those of categories or linear algebra while retaining completeness with respect to straightforward set-theoretic models.

However, these systems are of relatively little practical use because no automated reasoning tools, let alone efficient ones, are available. Therefore, our motivation is to translate one of these systems into first-order logic, FOL. Such a translation would translate a proof obligation to FOL and discharge it by calling existing FOL provers. This is called borrowing ([8]).

In principle, there are two ways how to establish the soundness of borrowing: proof-theoretically by translating the obtained proof back to the original logic, or model-theoretically by exhibiting a model-translation between the two logics. Proof-theoretical translations of languages with dependent types have been used in [9] to translate parts of DTT to simple type theory, in [10] to translate Mizar ([11]) into FOL, and in Scunak [12] to translate parts of DTT into FOL. Obtaining the FOL-proof for the proof-theoretical translation is possible in practice albeit somewhat tricky; but the back-translation of the proof is much

more difficult. In the cases of Mizar and Scunak, for instance, there is no proof translation specified or implemented.

Here we take here the model-theoretic approach and formulate a translation from the system introduced in [6] to FOL within the framework of institutions ([13]). Mathematically our main results can be summarized as follows. We use the institution DFOL as given in [6] and give an institution comorphism from DFOL into FOL. Every DFOL-signature is translated to a FOL-theory whose axioms are used to express the typing properties of the translated symbols. The signature translation uses an $n+1$-ary FOL-predicate P_s for every dependent type constructor s with n arguments. Then the formulas quantifying over x of type $s(t_1, \ldots, t_n)$ can be translated by relativizing (see [14]) using the predicate $P_s(t_1, \ldots, t_n, x)$.

More formally, we specify a functor Φ from the DFOL-signatures to FOL-theories. For each DFOL-signature Σ, we give a function α_Σ mapping DFOL-sentences over Σ to FOL-sentences over the translated signature $\Phi(\Sigma)$, and show that the family of functions α_Σ defines a natural transformation. Similarly, for each DFOL-signature Σ we give a functor β_Σ mapping FOL-models for the translated signature $\Phi(\Sigma)$ to DFOL-models for Σ, and show that the family of functors β_Σ defines a natural transformation. Finally, we prove the satisfaction condition for (Φ, α, β) and show that the comorphism admits model expansion. Our main theorem is the following:

Theorem 1. *(Φ, α, β) is an institution comorphism from DFOL to FOL that admits model expansion.*

Using the borrowing theorem ([8]), this yields the soundness of the translation, e.g., the problem of the theoremhood of DFOL-sentences can be reduced to the corresponding problem of FOL. DFOL provides a more natural way of formulating mathematical problems while staying close to FOL formally and intuitively. On the other hand, for FOL we have machine support in the form of automated theorem-provers and model-finders. The translation enables us to formulate a DFOL-problem, translate it to FOL, and then use the known automated methods to find a solution.

Thus, we provide a simple way to extend FOL theorem provers with dependently-typed input languages. It also becomes possible to integrate DFOL seamlessly into existing institution-based algebraic specification languages such as CASL ([15]). Finally, our result provides easier proofs of the free model and completeness theorems given in [6].

In the future we will integrate our translation into Hets ([16]), a CASL-based application that provides a framework for the implementation of institutions and institution translations. Since DFOL is defined within the Edinburgh Logical Framework (LF, [2]), we will also investigate how arbitrary institution and institution translation specifications in LF can be incorporated into Hets.

A full version of our treatment is available as [17].

References

1. Martin-Löf, P.: An Intuitionistic Theory of Types: Predicative Part. In: Proceedings of the Logic Colloquium 1973. (1975) 73–118
2. Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. Journal of the Association for Computing Machinery **40**(1) (1993) 143–184
3. Nordström, B., Petersson, K., Smith, J.: Programming in Martin-Löf's Type Theory: An Introduction. Oxford University Press (1990)
4. Pitts, A.: Categorical Logic. In Abramsky, S., Gabbay, D., Maibaum, T., eds.: Handbook of Logic in Computer Science, Volume 5. Algebraic and Logical Structures. Oxford University Press (2000) 39–128
5. Makkai, M.: First order logic with dependent sorts (FOLDS) (1997) Unpublished.
6. Rabe, F.: First-Order Logic with Dependent Types. In Shankar, N., Furbach, U., eds.: Proceedings of the 3rd International Joint Conference on Automated Reasoning. Volume 4130 of Lecture Notes in Computer Science., Springer (2006) 377–391
7. Belo, J.: Dependently Sorted Logic. In Miculan, M., Scagnetto, I., Honsell, F., eds.: TYPES 2008, Springer (2008) 33–50
8. Cerioli, M., Meseguer, J.: May I Borrow Your Logic? In Borzyszkowski, A., Sokolowski, S., eds.: Mathematical Foundations of Computer Science, Springer (1993) 342–351
9. Jacobs, B., Melham, T.: Translating dependent type theory into higher order logic. In Bezem, M., Groote, J., eds.: Typed Lambda Calculi and Applications. (1993) 209–29
10. Urban, J.: Translating Mizar for first-order theorem provers. In: MKM. (2003) 203–215
11. Trybulec, A., Blair, H.: Computer assisted reasoning with Mizar. In: Proceedings of the 9th International Joint Conference on Artificial Intelligence, Los Angeles, CA (1985) 26–28
12. Brown, C.: Combining Type Theory and Untyped Set Theory. In Shankar, N., Furbach, U., eds.: Proceedings of the 3rd International Joint Conference on Automated Reasoning, Springer (2006) 205–219
13. Goguen, J., Burstall, R.: Institutions: Abstract model theory for specification and programming. Journal of the Association for Computing Machinery **39(1)** (1992) 95–146
14. Oberschelp, A.: Untersuchungen zur mehrsortigen Quantorenlogik. Mathematische Annalen **145** (1962) 297–333
15. Astesiano, E., Bidoit, M., Kirchner, H., Krieg-Brückner, B., Mosses, P., Sannella, D., Tarlecki, A.: CASL: The Common Algebraic Specification Language. Theoretical Computer Science (2002)
16. Mossakowski, T., Maeder, C., Lüttich, K.: The Heterogeneous Tool Set. In O. Grumberg and M. Huth, ed.: TACAS 2007. Volume 4424 of Lecture Notes in Computer Science. (2007) 519–522
17. Sojakova, K., Rabe, F.: Translating dependently-typed logic to first-order logic (2008) Available at http://kwarc.info/frabe/dfol_fol.pdf.

Towards a Module System for K

Mark Hills and Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign, USA
201 N Goodwin Ave, Urbana, IL 61801
{mhills,grosu}@cs.uiuc.edu
http://fsl.cs.uiuc.edu

Abstract. To create reusable definitions of language features, isolated from changes to other features in the language, it is important that feature definitions be modular. This abstract introduces ongoing work on modularity features in K, an algebraic, rewriting logic based formalism for defining the semantics of programming languages.

Key words: language semantics, rewriting logic, modularity, K

One important aspect of formalisms for defining the semantics of programming languages is modularity. Modularity is generally expressed as the ability to add new language features, or modify existing features, without having to modify unrelated semantic rules. For instance, definitions using structural operational semantics (SOS) [1] are not modular: the standard change of adding a store to the configuration to allow for variable assignment requires adding a store to all SOS rules, even existing rules that do not need it. This is remedied in modular structural operational semantics (MSOS) [2, 3] by leveraging the labels on rule transitions to encode configuration elements, with the ability to elide unused parts of the configuration.

With a tool supported semantics, modularity can also be expressed as the ability to package language features into reusable units, which can then be assembled when defining a language. This form of modularity depends on the first: it should be possible to plug the same feature into multiple definitions, even in cases where (unused) parts of the configuration are different. Additionally, it should be possible to provide clean interfaces to language features and to different parts of the configuration, something not required in monolithic definitions.

This abstract provides a high-level overview of ongoing work on adding modularity features to K [4], an algebraic, rewriting logic based formalism for programming language semantics. This work is focused on both aspects of modularity mentioned above, allowing the packaging of language features for reuse while insulating existing features from unrelated changes to the language definition.

K Modules The module system in K is being designed to support a general module syntax incorporating the entire range of functionality needed when defining the semantics of a language, including the definition of abstract syntax, configuration items, and the semantics of language features. In theory, this would allow

a single, monolithic module to include definitions of all aspects of the semantics. However, to provide for a better separation of these constructs into more granular modules, and to allow for construct-specific defaults and syntax, specialized module formats for various constructs are being defined with a translation into the more general syntax. Initially, four different specialized module formats are being defined, for abstract syntax, configuration items, language features, and utility modules. Abstract syntax modules provide the abstract syntax for language constructs, and will (in the future) provide a point to connect transforms from concrete syntax. Configuration modules define elements of the configuration (state), such as the store, threads, environments, etc, and are intended to be assembled into the configuration used in a programming language definition. Language feature modules each define the semantics of a specific language feature. Since there can be multiple types of semantics – one for execution, one for type checking, and one for symbolic evaluation, for instance – this information can be noted in the feature definition, allowing the modular definition of multiple types of semantics across multiple language features. Finally, utility modules define general functionality which can be used in other modules.

Although small modules improve reuse, the large number of modules this leads to can make it challenging to work with language definitions, something noted in similar work on tool support for Action Semantics [5]. For K, work on tool support includes the ongoing development of an Eclipse plugin to provide a graphical environment for the creation and manipulation of K modules. This will initially include editor support, a graph view of module dependencies, and the ability to view both the language features used to define a language and the various semantics defined for a specific language feature. Longer-term goals include the graphical assembly of language configurations and links to an online database of reusable modules.

Context Transformers The rules used to define languages in rewriting logic often require additional context beyond just that used directly by the rule, especially in cases where the configuration is hierarchical. For instance, in many definitions the top level contains memory and threads; each thread contains an environment and control information; and control information includes the computation and potentially other items. An example of this, from the KOOL language [6], can be seen in Figure 1. A rule that uses the environment, computation, and store will need to mention the intervening parts of the configuration hierarchy, even though they are not inspected or modified; changes to these intervening parts can then require rule changes, breaking modularity. Context transformers provide a way to maintain modularity while still allowing hierarchical configuration definitions which vary across languages. Using context transformers, only those portions of the configuration actually used in a rule need to be mentioned. The transformers then transform the rule so that it will match the configuration hierarchy assembled for the language, with the intervening configuration items added automatically by the transformers.

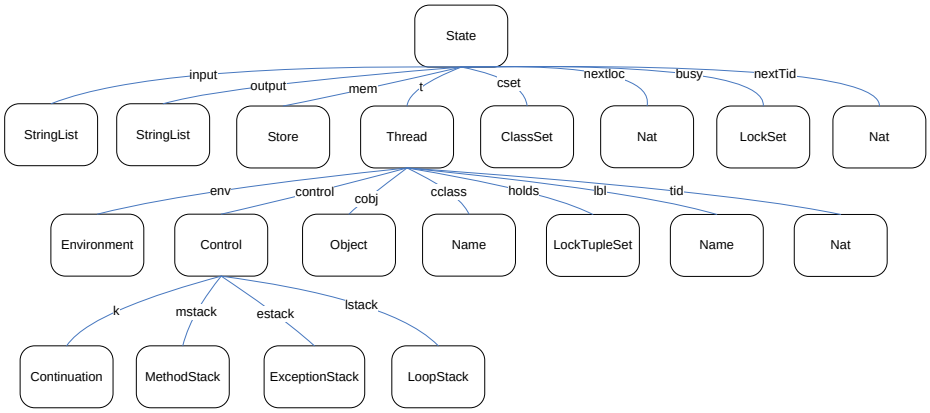


Fig. 1. KOOL State Infrastructure

Sort Inference and Variable Patterns Sort inference provides a way to automatically determine the sorts used by operators and assigned to variables, allowing definitions to be much more concise. Explicit sort annotations can be used in cases where the inferencer cannot determine the proper sort, or to document the expected sorts explicitly. Borrowing an idea from other systems [7], sets of variables can be defined based on patterns – for instance, by defining that variables starting with `Exp`, such as `Exp`, `Exp2`, `Exp3`, etc., are expressions.

References

1. Plotkin, G.D.: A Structural Approach to Operational Semantics. Journal of Logic and Algebraic Programming **60-61** (July-December 2004) 17–139
2. Mosses, P.D.: Foundations of Modular SOS. In: Proceedings of MFCS'99. Volume 1672 of LNCS., Springer (1999) 70–80
3. Mosses, P.D.: Pragmatics of Modular SOS. In: Proceedings of AMAST'02. Volume 2422 of LNCS., Springer (2002) 21–40
4. Roşu, G.: K: A Rewriting-Based Framework for Computations – Preliminary version. Technical Report UIUCDCS-R-2007-2926, University of Illinois at Urbana-Champaign (2007) Previous versions published as technical reports UIUCDCS-R-2006-2802 in December 2006, UIUCDCS-R-2005-2672 in 2005. K was first introduced in the context of Maude in Fall 2003 as part of a programming language design course (technical report UIUCDCS-R-2003-2897).
5. van den Brand, M., Iversen, J., Mosses, P.D.: An Action Environment. Science of Computer Programming **61**(3) (2006) 245–264
6. Hills, M., Roşu, G.: KOOL: An Application of Rewriting Logic to Language Prototyping and Analysis. In: Proceedings of RTA'07. Volume 4533 of LNCS., Springer (2007) 246–256
7. van den Brand, M., van Deursen, A., Heering, J., de Jong, H.A., de Jonge, M., Kuipers, T., Klint, P., Moonen, L., Olivier, P.A., Scheerder, J., Vinju, J.J., Visser, E., Visser, J.: The ASF+SDF Meta-environment: A Component-Based Language Development Environment. In: Proceedings of CC'01. Volume 2027 of LNCS., Springer (2001) 365–370

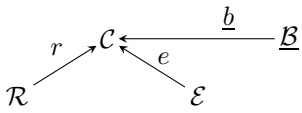
Architectures as Layered Graphs of Constructions*

Grzegorz Marczyński
gmarc@mimuw.edu.pl

Institute of Informatics, Warsaw University

Following the theoretical work of [1] where for a given indexed category we define the corresponding *indexed category of fragments*, in this paper we apply the concept of a fragment to the context of software architectures and their specifications. Given a semi-exact institution (cf. [2]) with a category of signatures being an indexed category flattened via the Grothendieck construction we define a category of *signature fragments*.

Our framework of specifications of system architectures has *a construction* as a basic architectural primitive. A construction signature is a simple sink diagram in the flattened category of signature fragments



where a signature $\mathcal{C}$ represents all symbols involved in the construction definition, $\mathcal{R}$ is a result signature, $\mathcal{E}$ is an external part signature, and finally $\underline{\mathcal{B}}$ is a signature fragment that represents the construction body. The body contains exactly these symbols that are defined by a construction and since it is a signature fragment, not a signature, it may contain e.g. function names even though it doesn't contain sort names used in their arities (see [1] for details). The only requirement we pose on such diagrams is that the pullback square of e and b be a pushout square (bicartesian square).

The representation of a construction body as a signature fragment allows the effective definition of a *fitting (span)* that connects two constructions provided the common part of their body signature fragments is an *empty fragment* (that is a signature fragment that may be not initial but essentially doesn't contain anything, cf. [1]). Two constructions connected by a fitting may be joined by a *sum* operation that corresponds to what is usually called an application of a generic unit (e.g. in CASL [3]). However, in contrast to the application, the sum, as we define it, is a symmetrical operation. Constructions and fittings give rise to the definition of a *graph of constructions*, i.e., a graph with nodes labeled by constructions and edges labeled by fittings. We lift the sum operation to work on graphs of constructions. Some other operations on constructions are also defined.

* This research was supported by the EC 6th Framework project SENSORIA: Software Engineering for Service-Oriented Overlay Computers, (IST-2005-016004).

If fittings are horizontal connections between construction signatures, *refinement morphisms* are vertical ones. A refinement morphism between two construction signatures consists of a tuple of morphisms between the corresponding signature parts, some of them being contravariant. Horizontal and vertical connections are used to define *layered graphs of constructions* where each layer is a graph of constructions with nodes labeled by construction signatures and edges labelled by fittings. Every node of a given layer may be vertically connected to a subgraph of a lower layer through a refinement morphism between the construction (a label of the node) and the sum of the constructions connected by fittings (labels of the subgraph). A vertical connection leaving a node labeled by a construction is called its *configuration*. Of course there may be more than one configuration for the construction. Each of them represents one possible architectural realization of the construction. A layered graph of constructions is called an *architecture* iff its highest level consists of exactly one node.

In the above description we mentioned only configuration signatures and definitions related to them. However, our work contains as well the definition of *construction models* that are partial mappings between models of $\mathcal{E}$ and $\mathcal{C}$ subject to persistency and compatibility conditions. We also define *construction specifications* that are pairs of specifications over the signatures $\mathcal{E}$ and $\mathcal{C}$. Moreover, we define all appropriate operations on construction models and specifications.

Finally we discuss the related work and show the advantages of our approach to architectural specification from CASL [3], SRML [4] and other architectural frameworks. We give some examples and present the prospects for the future work that among others contains the definition of an architectural logic to describe properties of layered refinement graphs in a declarative way.

References

1. Marczyński, G.: Indexed categories of fragments. (2008) Available at http://www.mimuw.edu.pl/~gmarc/papers/frag08.pdf.
2. Burstall, R.M., Goguen., J.A.: Institutions: Abstract model theory for specification and programming. Journal of the ACM **39**(1) (1992) 95–146
3. CoFi: CASL Reference Manual. Volume 2960 of Lecture Notes in Computer Science. Springer (2004)
4. Fiadeiro, J.L., Lopes, A., Bocchi, L.: Algebraic semantics of service component modules. In Fiadeiro, J.L., Schobbens, P.Y., eds.: WADT 2006. Volume 4409 of Lecture Notes in Computer Science., Springer (2007) 37–55

Integrating Formal Methods with Model-driven Engineering

Angelo Gargantini[1] Elvinia Riccobene[2] Patrizia Scandurra[2]

[1] Dip. di Ing. Informatica e Metodi Matematici, Università di Bergamo, Italy
angelo.gargantini@unibg.it
[2] Dip. di Tecnologie dell'Informazione, Università di Milano, Italy
{riccobene,scandurra}@dti.unimi.it

In this paper, we tackle the problem of integrating *formal methods* with the *Model-driven Engineering* (MDE) [1, 2] approach to software development.

It is widely acknowledged that the use of formal methods, based on rigourous mathematical foundations, is essential for system development, especially for high-integrity systems where safety or security are important. On the other hand, the MDE is emerging as new approach based on the systematic use of models as primary engineering artifacts throughout the engineering lifecycle by combining domain-specific modelling languages (DSMLs) with model transformers, analyzers, and generators. Both these two main approaches have advantages and disadvantages, which are briefly summarized in Fig. 1. We discuss here what advantages and disadvantages each approach has over the other, and illustrate how each can use its advantages to cover the disadvantages of the other. Specifically, we refer to our experience in integrating the Abstract State Machine (ASM) formal method [3] with MDE technologies.

The use of formal methods in system engineering is becoming essential, especially during the early phases of the development process. Indeed, an abstract model of the system can be used to understand if the system under development satisfies the given requirements (by simulation and model-based testing), and guarantees certain properties by formal analysis (validation & verification). While there are several cases proving the applicability of formal methods in industrial applications [4, 5] and showing very good results, many practitioners are, however, still reluctant to adopt formal methods. Besides the well-known lack of training, this skepticism is mainly due to: the complex notations that

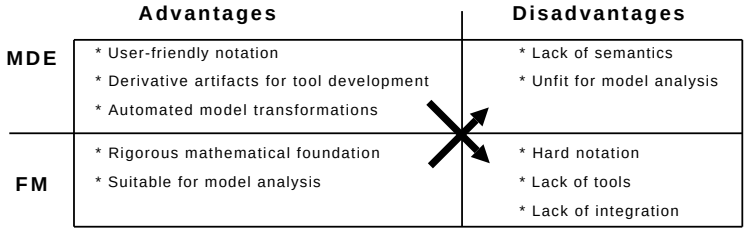


Fig. 1. Formal methods and MDE

formal techniques use rather than other lightweight and more intuitive graphical notations, like the Unified Modeling Language (UML); the lack of easy-to-use tools supporting a developer during the life cycle activities of the system development, possibly in a seamless manner; and the lack of integration among formal methods themselves and their associated tools.

MDE technologies with a greater focus on architecture and automation yield higher levels of abstraction in system development by promoting models as first-class artifacts to maintain, analyze, simulate, and eventually reduce into code or transformed into other models. Meta-modelling is a key concept of the MDE paradigm and it is intended as a way to endow a language or a formalism with an abstract notation, so separating the abstract syntax and semantics of the language from its different concrete notations. Metamodel-based modelling languages are increasingly being defined and adopted for specific domains of interest addressing the inability of third-generation languages to alleviate the complexity of platforms and express domain concepts effectively [2]. However, although the definition of a language abstract syntax by a metamodel is well mastered and supported by many meta-modelling environments (EMF/Ecore, GME/MetaGME, AMMA/KM3, XMF-Mosaic/Xcore, etc.), the semantics definition of this class of languages is currently an open and crucial issue. Currently, meta-modelling environments are able to cope well with most syntactic and transformation definition issues, but they lack of any standard and rigourous support to provide the (possibly executable) semantics of metamodels, which is usually given in natural language. This implies that most currently adopted metamodel-based languages are not yet suitable for effective formal analysis due to their lack of a strong semantics necessary for a formal model analysis assisted by tools.

Recently, we have been studying the feasibility of integrating the ASM formalism with MDE technologies (specifically, OMG/MDA and Eclipse/EMF). We started by defining a metamodel [6, 7], the Abstract State Machine Metamodel (AsmM), as abstract syntax description of a language for ASMs. From the AsmM, we obtained in a generative manner (i.e. semi-automatically) several artifacts (an interchange format, APIs, etc..) for the creation, storage, interchange, access and manipulation of ASM models. The AsmM and the combination of these language artifacts lead to an instantiation of the OMG metamodelling framework for the ASM application domain, the ASM mETA- modelling framework (ASMETA) that provides a global infrastructure for the interoperability of ASM tools (new and existing ones). The advantages of this work are twofold: ASMs can be used to provide semantics to languages defined in the MDE context, and MDE is useful in building and integrating tools around ASMs.

The lack of user-friendly notations, of integration of techniques, and of their tool inter-operability, is still a significant challenge for formal methods. Within the ASMs context, we developed an integrated set of tools [7, 8], based on the ASMETA framework, by exploiting the MDE metamodelling approach and its facilities (derivatives, libraries, APIs, etc.). On the basis of our experience in developing the ASMETA tool-set, we believe formal methods can gain benefits from the use of MDE automation means towards the integration of different formal

techniques and their tool inter-operability. These thoughts are also in sympathy with the *SRI Evidential Tool Bus idea* [9], and can contribute positively to examine inter-operability issues and solutions that can forge the beginning of a new era in the analysis and development of computer systems and software.

We have been addressing the issue of defining a formal *semantic framework* based on the ASM formal method to provide languages with their semantics natively with their metamodels. We have been applying [10] the proposed methodology to the OMG metamodelling framework for the behaviour specification of state-like formalisms like the Finite State Machines, Petri nets, etc., provided in terms of a metamodel, and, as major case study, for the semantics definition of the *SystemC UML profile* [11] – an extension of the UML for high-level modelling of embedded systems.

The definition of a means for specifying rigorously the semantics of metamodels is a necessary step in order to develop formal analysis techniques and tools in the model-driven context. Along this research line, we are also tackling the problem of formally analysing visual models developed with the UML Profile for SystemC [12]. We believe MDE principles and technologies combined with formal methods elevate the total level of automation possible and provide the widely demanded formal analysis support.

References

1. Bézivin, J.: On the Unification Power of Models. Software and System Modeling **4**(2) (2005) 171–188
2. Schmidt, D.C.: Guest editor's introduction: Model-driven engineering. IEEE Computer **39**(2) (2006) 25–31
3. Börger, E., Stärk, R.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer Verlag (2003)
4. Formal Methods Europe: `http://www.fmeurope.org`
5. Formal Methods Virtual Library: `http://www.afm.sbu.ac.uk/`
6. Gargantini, A., Riccobene, E., Scandurra, P.: Metamodelling a Formal Method: Applying MDE to Abstract State Machines. Tech. Rep. 97, DTI Dept., University of Milan (2006)
7. The Abstract State Machine Metamodel: `http://asmeta.sf.net/` (2006)
8. Gargantini, A., Riccobene, E., Scandurra, P.: A Language and a Simulation Engine for Abstract State Machines based on Metamodelling. J. of Universal Computer Science ((To appear) 2008)
9. Rushby, J.M.: Harnessing disruptive innovation in formal verification. In: SEFM. (2006) 21–30
10. Gargantini, A., Riccobene, E., Scandurra, P.: A precise and executable semantics of the SystemC UML profile by the meta-hooking approach. Technical Report 110, DTI Dept., University of Milan (2008)
11. Riccobene, E., Scandurra, P., Rosti, A., Bocchio, S.: A UML 2.0 profile for SystemC: toward high-level SoC design. In: EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software, ACM (2005) 138–141
12. Gargantini, A., Riccobene, E., Scandurra, P.: A Model-driven Validation & Verification Environment for Embedded Systems. In: Proc. of the IEEE third Symposium on Industrial Embedded Systems (SIES'08), IEEE (2008)

Distributed Specifications in Heterogeneous Logical Environments* (Preliminary Abstract)

Andrzej Tarlecki**

Institute of Informatics, Warsaw University
and Institute of Computer Science PAS, Warsaw, Poland.

The theory of *institutions* [4] provides an excellent framework where the theory of specification and formal software development may be presented in an adequately general and abstract way [5–7]. The initial work within this area captured specifications built and developments carried out in an arbitrary but fixed logical system formalised as an institution. However, the practise of software specification and development goes much beyond this. Different logical systems may be appropriate or most convenient for specification of different modules of the same system, of different aspects of system behaviour, or of different stages of system development. This leads to the need for a number of logical systems to be used in the same specification and development project. This observation spurred a substantial amount of research work already, and motivates the research to be presented here.

As before, we formalise all the logical systems one may potentially use in a project as institutions. Moreover, to enable a sensible use of a number of such institutions together, they must be linked with each other in some semantically meaningful way. Such links were originally formalised as institution morphisms [4], but then a number of other concepts of a map between institutions emerged, as systematically presented in [8], and have been put to use to move between institutions in various ways with various purposes in mind [9]. In particular, institution morphisms and comorphisms [10] offer natural ways to move specifications from one institution to another. Using them together with other standard (intra-institutional) specification-building operations, one constructs *heterogeneous structured specifications* [11]. Here, heterogeneous specifications may involve a number of institutions to specify some aspects or some parts of the system, but ultimately are focused at one institution of interest, where the overall specification ends up, specifying programs as captured by models of this institutions.

In such a framework, one works in a *heterogeneous logical environment* formed by a number of logical systems formalised as institutions and linked with each other in a way captured by various maps between institutions. One such logical environment is the HETS family of institutions [12], supported by a tool to build and work with heterogeneous specifications [13].

* This work was funded in part by the European IST FET programme under the IST-2005-015905 MOBIUS and IST-2005-016004 SENSORIA projects.

** Based on joint work with Till Mossakowski [1], with María Victoria Cengarle, Alexander Knapp and Martin Wirsing [2], and with Adam Warski [3].

One issue that we want to discuss in detail is that each basic concept of a map between institutions can be captured by institution (co)morphisms — as a span of (co)morphisms [1]. Replacing a map between institutions by an appropriate span of comorphisms allows one to represent exactly the same relationship between institutions and their components; technically, no information is lost. Consequently, in principle any heterogeneous logical environment may be represented as a diagram in the category of institutions with institution comorphisms. We discuss to what extent this is also appropriate from the methodological and pragmatic point of view. Moreover, such diagrams may be combined in various ways, using (co)limits in the appropriate categories of institutions, or by the Grothendieck construction that puts all the component institutions side by side in a single "heterogeneous" institution [14]. We argue that even in cases where this brings no loss of technical content, such "flattening" of heterogeneous logical environments is not methodologically desirable.

Another family of logical systems is suggested by UML [15], where system specifications typically involve a number of diagrams of different kinds, each capturing a different aspect of the system. Each kind of UML diagrams leads to a separate logical system which, at least in principle, can be formalised as an institution. Expected relationships between system properties specified by different kinds of UML diagrams may now be captured using appropriate institution maps [2]. UML specifications, however, are quite different from heterogeneous specifications mentioned above. They do not focus at any single UML institution, but rather form a collection of specifications residing in different institutions of UML diagrams. Explaining how such a collection is to be interpreted in the particular heterogeneous logical environment of UML is essential for understanding of the semantics of UML specifications [2].

We present a general abstract concept of a *distributed heterogeneous specification*. Such distributed heterogeneous specifications consist of a collection of specifications focused at various institutions in an underlying heterogeneous logical environment, linked by specification morphisms generalised by involving institution maps. If the environment is uniform in the sense that all the links between institutions are captured by (spans of) comorphisms, then such a distributed specification is simply a diagram in the category of specifications and their inter-institutional morphisms built over this environment. We discuss to what extent such uniformity is achievable and desirable from the methodological and pragmatic point of view.

Distributed heterogeneous specifications come equipped with a rather natural semantics, given in terms of compatible families of models of component specifications. As in [5, 16] this yields in the standard way a number of usual concepts: consistency, semantic consequence, and perhaps most importantly, implementation of one distributed specification by another. We show some typical cases of such implementation steps, and stress how the structure of distributed specification may evolve in the course of development.

References

1. Mossakowski, T., Tarlecki, A.: Heterogeneous specification. Universität Bremen, in preparation (2008)
2. Cengarle, M., Knapp, A., Tarlecki, A., Wirsing, M.: A heterogeneous approach to UML semantics. In Degano, P., Nicola, R.D., Meseguer, J., eds.: Festschrift in Honour of Ugo Montanari's 65ht Birthday. LNCS. Springer (2008) to appear.
3. Warski, A.: Limits and colimits in categories of institutions. In Haveraaen, M., Power, J., Seisenberger, M., eds.: CALCO Young Researchers Workshop, CALCO-jnr 2007. (2007)
4. Goguen, J., Burstall, R.: Institutions: Abstract model theory for specification and programming. Journal of the ACM **39**(1) (1992) 95–146
5. Sannella, D., Tarlecki, A.: Specifications in an arbitrary institution. Information and Computation **76** (1988) 165–210
6. Sannella, D., Tarlecki, A.: Essential concepts of algebraic specification and program development. Formal Aspects of Computing **9** (1997) 229–269
7. Tarlecki, A.: Abstract specification theory: An overview. In Broy, M., Pizka, M., eds.: Models, Algebras, and Logics of Engineering Software. Volume 191 of NATO Science Series — Computer and System Sciences. IOS Press (2003) 43–79
8. Goguen, J., Rosu, G.: Institution morphisms. Formal Aspects of Compututing **13**(3-5) (2002) 274–307
9. Tarlecki, A.: Moving between logical systems. In Haveraaen, M., Dahl, O.J., Owe, O., eds.: Recent Trends in Data Type Specifications. Selected Papers, 11th Workshop on Specification of Abstract Data Types ADT'95. LNCS 1130, Springer (1996) 478–502
10. Meseguer, J.: General logics. In: Logic Colloquium 87. North Holland (1989) 275–329
11. Tarlecki, A.: Towards heterogeneous specifications. In Gabbay, D., de Rijke, M., eds.: Frontiers of Combining Systems 2. Studies in Logic and Computation. Research Studies Press (2000) 337–360
12. Mossakowski, T.: Heterogeneous Specification and the Heterogeneous Tool Set. Habilitation thesis, Universität Bremen (2005)
13. Mossakowski, T., Maeder, C., Lüttich, K.: The Heterogeneous Tool Set. In Grumberg, O., Huth, M., eds.: TACAS 2007. LNCS 4424, Springer (2007) 519–522
14. Diaconescu, R.: Grothendieck institutions. J. Applied Categorical Structures **10** (2002) 383–402
15. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. Addison-Wesley (1998)
16. Sannella, D., Tarlecki, A.: Toward formal development of programs from algebraic specifications: Implementations revisited. Acta Informatica **25** (1988) 233–281

Generalized theoroidal institution comorphisms

Mihai Codescu[1] and Till Mossakowski[1,2]

[1] DFKI Laboratory, Bremen
[2] Department of Computer Science, University of Bremen

Institution comorphisms abstractly capture the notion of logical embedding or encoding. Theoroidal institution comorphisms [1], called maps of institutions in [2], map theories of the source institution to theories of the target institution in such a way that the signature is preserved (i.e. theories over the same signature are mapped to theories over the same signature). However, when practically implementing logics and comorphisms in the Heterogeneous Tool Set[3], in several situations, we encountered logic translations of practical interest that do not have this property.

For example, in CASL the free types are axioms, while in Isabelle they are part of the signatures, so the signature changes in the presence of the free type axioms. Another translation which fails to be a comorphism is the encoding of partiality in CASL with the help of a definedness predicate, as defined in [4], but this time in the presence of subsorting. There, a 'bottom' constant is added on each sort to model undefined elements, and, whenever a sentence has casts to supersorts or prejections to the subsorts, these 'bottom' constants occur on the subsort as well.

The formalization of logics and logic translations as specification frames and specification frame comorphisms [5] does not make any assumption about the mapping of theories, but the concept of specification frame lacks the notion of sentence and satisfaction and in such a setting it would be much more difficult, if doable, to provide proof support and to integrate external tools.

Since the definition of specification frames regards theories simply as objects of some fixed category and does not impose any restriction on their structure, we can use them to store data about sentences. Namely, given a institution I, we can derive from it a specification frame $SF(I)$ whose theories are triples consisting of a signature in I and two sets of Σ-sentences, one for axioms and the other for goals [6]. This separation allows us to keep distinction between axioms and proof goals via this derivation.

Then, given two institutions I and I', a generalized institution comorphism from I to I' is a specification frame comorphism from $SF(I)$ to $SF(I')$.

The next step is to define a Grothendieck construction over a graph of logic and their translations, this time modeled as institutions and generalized comorphisms. The result of this construction is itself a specification frame with the same particular shape of theories. A theory morphism in the Grothendieck frame consists of a translation along a generalized institution comorphism, followed by a intra-institution morphism from the translated theory to the target one.

A canonical way to add sentences to specification frames is to consider as sentences over a theory T all presentation morphisms of source T. The intuition

behind this is that a sentence is added to the set of goals, and one gets thus a theory inclusion. In order to properly define the sentences translations along signature morphisms, we need to have a canonical selection of pushouts.

Given a theory T, a model of the theory satisfies a sentence (i.e. a theory morphism ϕ of source T) if it can be expanded along ϕ to a model of the target theory.

Note that after adding the sentences and the satisfaction relation, we fail to obtain an institution, because, in order to have satisfaction preserved by expansions, we would need all logics from our logic graph to be weakly semi-exact and all generalized comorphisms to be weakly exact. However, the other implication of the satisfaction condition holds, so we get a so-called *rps-institution* [7].

We investigate how this new framework is suited as a foundation for heterogeneous specifications and also discuss its implementation in the Heterogeneous Tool Set. The most difficult thing is to add an entailment system to the rps-institution, by using the formalism of development graphs [8], introduced as a tool for structured theorem proving and proof management. In [9] the development graph calculus was adapted to the heterogeneous framework and proved complete w.r.t. a given oracle for conservative extensions. We prove that most of the rules of the development graph calculus remain sound in the new setting and argue that we can regard the unsound ones as only introduced for optimization purposes. The completeness is however lost.

References

1. Goguen, J., Roşu, G.: Institution morphisms. Formal Aspects of Computing **13** (2002) 274–307
2. Meseguer, J.: General logics. In Ebbinghaus, H.D., et al., eds.: Proceedings, Logic Colloquium, 1987. North-Holland (1989) 275–329
3. Mossakowski, T., Maeder, C., Lüttich, K.: The Heterogeneous Tool Set. In Grumberg, O., Huth, M., eds.: TACAS 2007. Volume 4424 of Lecture Notes in Computer Science., Springer-Verlag Heidelberg (2007) 519–522
4. Mossakowski, T.: Relating CASL with other specification languages: the institution level. Theoretical Computer Science **286** (2002) 367–475
5. Ehrig, H., Pepper, P., Orejas, F.: On recent trends in algebraic specification. In Ausiello, G., Dezani-Ciancaglini, M., Rocca, S.R.D., eds.: ICALP. Volume 372 of Lecture Notes in Computer Science., Springer (1989) 263–288
6. Mossakowski, T.: Representations, hierarchies and graphs of institutions. PhD thesis, Universität Bremen; www.uni-bremen.de (1996) Also appeared as book in Logos Verlag.
7. Salibra, A., Scollo, G.: A soft stairway to institutions. In Bidoit, M., Choppy, C., eds.: COMPASS/ADT. Volume 655 of Lecture Notes in Computer Science., Springer (1991) 310–329
8. Mossakowski, T., Autexier, S., Hutter, D.: Development graphs – proof management for structured specifications. Journal of Logic and Algebraic Programming **67**(1-2) (2006) 114–145
9. Mossakowski, T.: Heterogeneous specification and the heterogeneous tool set. Habilitation thesis, University of Bremen (2005)

Heterogeneous Model Finding with HETS

Dominik Lücke[1] and Till Mossakowski[1,2]

[1] SFB/TR 8 Spatial Cognition
University of Bremen
[2] DFKI Lab Bremen

When designing logical theories in a heterogeneous setting in a modular way, as it is supported in the Heterogeneous Tool Set HETS [1], it is of great importance that all specifications involved are consistent (= have at least a model) and to find at least a guard for those models. HETS uses the following approaches to check consistency of CASL specifications:

1. a tool similar to the CASL consistency checker [2]. It can check consistency of specifications of datatypes with recursively defined functions by just checking certain syntactic restrictions;
2. the resolution prover `SPASS` [3] can derive consistency via reaching a saturated set of clauses. This works in principle for arbitrary first-order specifications, but in practice it works well only in few cases, and it does not output a model;
3. `Isabelle-refute` [4] uses a SAT solver to find finite models of arbitrary first-order specifications; however, these are of rather limited size; Isabelle can use its internal SAT solver based on the DPLL procedure or be connected to external tools like `zChaff` [5] and `minisat` [6];
4. the first order theorem prover `Darwin` [7] is able to find (also somewhat larger) finite models of arbitrary first-order specifications. Darwin implements the Model-Evolution-Calculus [8], which is roughly a lifting of the well known DPLL-Algorithm to first order logic.

A very interesting feature of `Darwin` is that it can output models in `TPTP` and other concrete syntaxes for logics. This enables us to send a CASL specification (via a translation to untyped first-order logic) to `Darwin`, let it find a model (if there is one) and to translate the output back to CASL, to present the user of HETS the model in a convenient way. The integration of this feature is currently ongoing work.

This further leads to the question of how to represent CASL models. Of course, we cannot expect to represent all models in a finite way, but we can at least represent finite models and some infinite term generated models. We design a sublanguage of CASL specifications that are guaranteed (by syntactic restriction) to be consistent, and use this as representations for models. By further allowing the reduction of a model along a signature morphism, we can also represent models that are not term generated. Hence, a Σ-model is represented by a triple $(\Sigma_1, \Gamma_1, \sigma)$, where (Σ_1, Γ_1) is a basic specification satisfying certain restrictions, and $\sigma : \Sigma \to \Sigma_1$ is a signature morphism. We further can distinguish between definitional, monomorphic, and arbitrary model representations,

where a definitional representation represents a unique model (this only works for the propositional fragment of CASL), a monomorphic one a model unique up to isomorphism, and an arbitrary representation represents a class of models.

This opens the perspective to generalise these model representations to other institutions, such that models become first-class citizens in the Heterogeneous Tool Set HETS, and can be reduced not only against signature morphisms, but also against institution comorphisms [9].

Finally, `QuickCheck` is a model-checker for CASL directly integrated into HETS. It is inspired by the equally named testing tool [10] for Haskell programs. It takes a model representation and evaluates a formula in the represented model or model class. `QuickCheck` has been successfully used in the process of finding a model for the DOLCE Ontology [11], which has proceeded by writing a view from the specification to a hand-crafted (representation of a) model. This is indeed still another method for model finding.

The use of `QuickCheck` leads us to an approach that one could call assisted model finding. In our heterogeneous approach, automatic model finders (such as `Darwin`) can be used to search for models in a structured specification for as many theory as possible—this means as long as the model finder is able to find something in reasonable time. When this strategy is exhausted, one can table the models that were generated by the model finders and use these to build models for more complicated specifications by hand. We have made some experiments with this on the first-order formulation of the DOLCE ontology. During that work we had to find out that in this special case `Isabelle-refute` augmented by `zChaff` and `minisat` surprisingly performed far better than `Darwin`. It was even possible to find models of specifications with Isabelle where `Darwin` already gave up. But during further experiments, we had to find out that `Isabelle-refute` quite quickly ran into problems when the CASL subsorting feature was used extensively. `Isabelle` does not support subsorting as in CASL, and the coding of the CASL subsorting to `Isabelle` led to an inefficiency that led to a blow-up in the memory utilization of that tool.

Currently we are evaluating some more model finders for the integration into HETS. We are especially having a look at `Paradox` [12] and `Mace4` [13]. With these model finders we made some experiments with the first-order formulation of DOLCE, too. We had to find out that they do not perform any better than `Darwin` on the tasks that we gave to them. On the positive side, specifications in `TPTP` can be used with these tools, as well as with `Darwin`, making the integration into HETS rather easy.

Acknowledgements

Work on this paper has been supported by the DFG-funded collaborative research center SFB/TR 8 'Spatial Cognition' and by the German Federal Ministry of Education and Research (Project 01 IW 07002 FormalSafe). We thank Oliver Kutz and Christian Maeder for fruitful discussions and Erwin R. Catesbeiana for pointing out some cases where model finding is fruitless.

References

1. Mossakowski, T., Maeder, C., Lüttich, K.: The Heterogeneous Tool Set. In Grumberg, O., Huth, M., eds.: TACAS 2007. Volume 4424 of Lecture Notes in Computer Science., Springer-Verlag Heidelberg (2007) 519–522
2. Lüth, C., Roggenbach, M., Schröder, L.: CCC - the casl consistency checker. In Fiadeiro, J., ed.: Recent Trends in Algebraic Development Techniques, 17th International Workshop (WADT 2004). Volume 3423 of Lecture Notes in Computer Science., Springer; Berlin (2005) 94–105
3. Weidenbach, C., Brahm, U., Hillenbrand, T., Keen, E., Theobalt, C., Topic, D.: SPASS version 2.0. In Voronkov, A., ed.: Automated Deduction – CADE-18. Volume 2392 of LNCS ., Springer Verlag; Berlin (2002) 275–279
4. Weber, T.: Bounded model generation for Isabelle/HOL. In: Electronic Notes in Theoretical Computer Science. Volume 125. (2005) 103–116
5. Mahajan, Y., Fu, Z., Malik, S.: Zchaff2004: An Efficient SAT Solver. In: Theory and Applications of Satisfiability Testing. Volume 3542., Springer (2005) 360–375
6. Een, N., Srensson, N.: An extensible sat-solver
7. Baumgartner, P., Fuchs, A., Tinelli, C.: Darwin: A Theorem Prover for the Model Evolution Calculus. In Schulz, S., Sutcliffe, G., Tammet, T., eds.: IJCAR Workshop on Empirically Successful First Order Reasoning (ESFOR (aka S4)). Electronic Notes in Theoretical Computer Science (2004)
8. Baumgartner, P., Tinelli, C.: The Model Evolution Calculus. In Baader, F., ed.: CADE-19 – The 19th International Conference on Automated Deduction. Volume 2741 of Lecture Notes in Artificial Intelligence., Springer (2003) 350–364
9. Goguen, J., Roşu, G.: Institution morphisms. Formal aspects of computing **13** (2002) 274–307
10. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: International Conference on Functional Programming. Volume 35 of ACM Sigplan Notices. (2000) 268–279
11. Gangemi, A., Guarino, N., Masolo, C., Oltramari, A., Schneider, L..: Sweetening Ontologies with DOLCE. In Gómez-Pérez, A., Benjamins, V.R., eds.: Knowledge Engineering and Knowledge Management. Ontologies an d the Semantic Web, 13th International Conference, EKAW 2002, Sigue nza, Spain, October 1-4, 2002, Proceedings. LNCS 2473, Springer Verlag; Berlin (2002) 166–181
12. Claessen, K., Sörensson, N.: New Techniques that Improve MACE-style Model Finding . In: Proc. of Workshop on Model Computation (MODEL). (2003)
13. McCune, W.: Mace4 Reference Manual and Guide. Technical report, Memo ANL/MCS-TM-264, Mathematics and Computer Science Division, Argonne National Laboratory (2003)

Monitoring Java Code Using ConGu

Vasco T. Vasconcelos, Isabel Nunes, and Antónia Lopes

Faculty of Sciences of the University of Lisbon, Campo Grande, 1749–016 Lisboa, Portugal,
{vv,in,mal}@di.fc.ul.pt

The formal specification of software components is an important activity within the task of software development, insofar as formal specifications are useful, on the one hand, to understand and reuse software and, on the other hand, to test implementations for correctness.

Design by Contract (DBC) [1] is widely used for the specification of object-oriented software. There are a number of languages and tools (e.g., [2,3,4,5]) that allow equipping classes and methods with invariants, pre and post-conditions, which can be monitored for violations at runtime. In the DBC approach, specifications are class interfaces (Java interfaces, Eiffel abstract classes, etc) annotated with contracts expressed in a particular assertion language, which is usually an extension of the language of boolean expressions of the OO language.

To build contracts using these languages one must observe the following: *(i)* contracts are built from boolean assertions, thus procedures (methods that do not return values) cannot be used; *(ii)* contracts should refer only to the public features of the class because client classes must be able not only to understand contracts, but also to invoke operations that are referred to in them—e.g., clients must be able to test pre-conditions; *(iii)* to be monitorable, a contract cannot have side effects, thus it cannot invoke methods that modify the state. These restrictions bring severe limitations to the kind of properties we can express directly through contracts. Unless we define a number of, otherwise dispensable, additional methods, we are left with very poor specifications.

Model-based approaches to DBC, like those proposed for Z [6], Larch [7], JML [5], and AsmL [8], overcome these limitations by specifying the behavior of a class, not via the methods available in the class, but else through very abstract implementations based on basic elements available in the adopted specification language. Rather than a *model based* approach, we instead adopted a *property based* algebraic approach to specifications, motivated and described in reference [9].

ConGu (Contract Guided System Development [10]) is a project whose aim is the development of a framework to create property-driven algebraic specifications and to fully test Java implementations against them. We find it important to equip property-driven approaches with tools similar to the ones currently available for model-driven approaches. Support for checking implementations against algebraic specifications is, as far as we know, restricted to a few approaches (cf [11,12]), which have limitations our approach overcomes.

The key idea of the **ConGu** approach is to reduce the problem of testing implementations against algebraic specifications to the runtime monitoring of contract annotated classes, which are automatically generated. Runtime contract monitoring is supported today by several runtime assertion-checking tools.

The **ConGu** main components are specifications, modules, and refinements. The *specifications* we use in this context are algebraic, property-driven insofar as they define sorts and operations on those sorts. In general terms, **ConGu** supports partial specifications—whose operations can be interpreted by partial functions—with conditional axioms. Each specification defines a single sort but it may use other sorts while defining, for example, parameters or results of operations. Specifications with external references to other sorts or operations are meaningful only when they are put together with the specifications that define all those references. We use the notion of *module* to denote the set of specifications that, together, are self-contained.

In order to check the behavior of Java classes against specifications—violations of an axiom or a domain restriction—the gap between specifications and Java classes must be bridged. For this purpose, *refinement mappings* have to be defined indicating which sort is implemented by which class, and which operation is implemented by which method. Because this activity does not require any knowledge about the concrete representation, refinement mappings are quite simple to define.

In this presentation we put forward an overview of the **ConGu** framework and demonstrate the **ConGu** tool, implemented as a plugin for the Eclipse IDE. The tool allows users to test Java classes—no source code needed, just bytecode—against a module of specifications, and to discover runtime axiom violations. It reads algebraic specifications and a mapping relating specifications and Java entities, and generates a number of classes that are used to test the original implementation against the given specifications, in a way that is transparent to the user. The technique used by **ConGu** surpasses the above referred limitations in what contracts are concerned: all specification properties are checked against implementations because monitorable contracts are generated that cover them all. We also report on the use of the **ConGu** tool in the context of an undergraduate programming course.

References

1. Meyer, B.: Object-Oriented Software Construction. 2nd edn. Prentice-Hall PTR (1997)
2. Barnett, M., Leino, K.R.M., Schulte, W.: The spec# programming system: An overview. In: Proc. of CASSIS 2004. Number 3362 in LNCS, Springer (2004)
3. Bartetzko, D., Fisher, C., Moller, M., Wehrheim, H.: Jass - Java with assertions. ENTCS **55**(2) (2001)
4. Henne-Wu, R., Mitchell, W., Zhang, C.: Support for design by contract in the C# programming language. Journal of Object Technology **4**(7) (2004) 65–82
5. Java Modelling Language: `http://www.jmlspecs.org/`
6. Spivey, J.: The Z Notation: A Reference Manual. ISCS. Prentice-Hall (1992)
7. Guttag, J., Horning, J., Garland, S., Jones, K., Modet, A., Wing, J.: Larch: Languages and Tools for Formal Specification. Springer (1993)
8. Barnett, M., Schulte, W.: Spying on components: A runtime verification technique. In: Proc. WSVCBS — OOPSLA 2001. (2001)
9. Nunes, I., Lopes, A., Vasconcelos, V.T., Abreu, J., Reis, L.S.: Checking the conformance of Java classes against algebraic specifications. In: Proceedings of ICFEM'06. Volume 4260 of LNCS., Springer-Verlag (2006) 494–513
10. Congu: Monitoring Java Code Against Algebraic Specifications: `http://gloss.di.fc.ul.pt/congu/`

11. Antoy, S., Hamlet, R.: Automatically checking an implementation against its formal specification. IEEE TOSE **26**(1) (2000) 55–69
12. Henkel, J., Diwan, A.: A tool for writing and debugging algebraic specifications. In: Proc. ICSE 2004. (2004)

Transformations of Conditional Rewrite Systems Revisited (Extended Abstract)

Karl Gmeiner and Bernhard Gramlich

TU Wien, Austria, {gmeiner,gramlich}@logic.at

We revisit known transformations of conditional rewrite systems to unconditional ones in a systematic way. We present a unified framework for describing and classifying such transformations, discuss the major problems arising, provide simplified (old) and new counterexamples to certain (desirable) properties of specific transformations, and finally present a new transformation which has some advantages as compared to a quite recent approach, namely the one of [1].[1] In this abstract, due to lack of space we focus on the latter contribution, after briefly discussing major general issues with such transformation approaches.

Conditional term rewrite systems (CTRSs) and conditional equational specifications are very important in algebraic specification, prototyping, implementation and programming. They naturally occur in most practical applications. Yet, compared to unconditional term rewrite systems (TRSs), CTRSs are much more complicated, both in theory (especially concerning criteria and proof techniques for major properties of such systems like confluence and termination) and practice (implementing conditional rewriting in a clever way is far from being obvious, due to the inherent recursion when evaluating conditions). For these (theoretical and practical) reasons, transforming CTRSs into (unconditional) TRSs in an adequate way has been studied for a long time cf. e.g. [3], [4], [5], [6], [7], [8], [9], [1], [10], [11], [12]. The motivations for these transformations were manifold, depending on the overall goal of the analysis (see below).

Roughly, all transformations work by translating the original syntax (signature and terms) into an extended or modified one using auxiliary function symbols, and by translating the rules in a corresponding way such that the evaluation of conditions and some control structure is (appropriately) encoded within the resulting unconditional TRS.

Already from this abstract point of view, the main questions and problems of such transformation approaches can be inferred:

- How are the relevant (syntactical and semantic) properties of a given CTRS $\mathcal{R}$ and its transformed TRS $\mathcal{R}'$ related?
- Is it possible to infer a property $P(\mathcal{R})$ from $P(\mathcal{R}')$ (*soundness*) and vice versa (*completeness*). Typically, and unlike in many other settings, here *completeness* properties are less difficult to show/obtain than *soundness* properties. The intuitive reason is that the transformations are designed such that every reduction step that was possible in the original system can be simulated

[1] The work presented here is partially based on [2].

by a reduction (sequence) in the transformed system. On the other hand, since the evaluation of conditions in the transformed system is done using unconditional rewrite rules, there is no obvious encapsulation any more as in the conditional case. And this entails the danger of enabling reductions that were originally impossible (because a failed attempt to verify a condition in the conditional system has no further consequences). Hence, for instance soundness usually is a problem.
- From a theoretical point of view: Is a given transformation useful for analyzing a given CTRS via its transformed unconditional version?
- From a practical point of view: Does a given transformation yield an operational / executable specification / high-level implementation with good properties, e.g. in terms of the input/output behaviour, of efficiency, of comprehensibility, ...? Is (explicit meta-level) backtracking, corresponding to failed attempts of verifying conditions, in the transformed system avoided?

One property of transformations which is particularly important from a practical point of view, is the following: If we start a simulation (a reduction in the transformed TRS) from an transformed initial term and obtain a normal form in the transformed system, then the latter should correspond to a normal form of the initial term in the original CTRS (this property, together with a few other requirements, is called *computational equivalence* in [1]). Otherwise, some form of backtracking would be needed, because then we are stuck with a failed attempt of verifying conditions. As an example consider the (*oriented normal*) CTRS $\mathcal{R}$ (cf. [8, 1]) consisting of the two rules

$$f(g(x)) \to 0 \;\Leftarrow\; x \to 0 \qquad g(g(x)) \to g(x)$$

and the initial term $t = f(g(g(0)))$. *Unravelings* following the approach of [6, 7] use new function symbols to encode conditions and store the variable bindings until finally – if the conditions are verified – the right-hand side may be produced:

$$f(g(x)) \to U_1(x, x) \quad U_1(0, x) \to 0 \quad g(g(x)) \to g(x)$$

Here, t reduces to normal forms 0 and $U_1(g(0), g(0))$. However, the latter term does not correspond to a normal form in the original CTRS.

Transformations like the one of [8] increase the arity of some function symbols and encode the conditions in these new "conditional" arguments. For notational simplicity we will collect them in lists denoted by $[\ldots]$ where $[\,]$ represents the empty list. Using this approach, we get:

$$f'(g(x), [\,]) \to f'(g(x), [x]) \quad f'(g(x), [0]) \to 0 \quad g(g(x)) \to g(x)\,.$$

For *constructor–based* CTRSs, this approach of [8] yields good results, but in our example, which is not a constructor system, we still obtain – from the transformed initial term $t' = f'(g(g(0)), [\,])$ – an undesired normal form $f'(g(0), [g(0)])$ that does not correspond to a normal form of the initial term t in the original CTRS.

To solve this problem, [1] proposed an additional unary operand $\{.\}$ for "resetting" conditional arguments whenever an "inner" rewrite step occurs:

$$\begin{array}{ccccc} f'(g(x),[\,]) \to f'(g(x),\{x\}) & & f'(g(x),\{0\}) \to \{0\} & & g(g(x)) \to \{g(x)\} \\ f'(\{x\},z) \to \{f'(x,[\,])\} & & g(\{x\}) \to \{g(x)\} & & \{\{x\}\} \to \{x\} \end{array}$$

Here, the only normal form of the transformed initial term $t' = f'(g(g(0)),[\,])$ is $\{0\}$ which corresponds to 0 as desired. Yet, in general this transformation destroys some syntactical properties of the original CTRS like being a constructor system, reinforces sequential processing of conditions and "(too) often" resets encoded conditions.

In our approach we encode conditions at "appropriate more inner" positions such that propagation of "reset information" is earlier possible. This approach also works for CTRSs with deterministic extra variables (DCTRSs), is a proper extension of the transformation of [8] and has better support for parallel rewriting. In the example, our new transformation yields the TRS

$$f(g'(x,[\,])) \to f(g'(x,[x])) \quad f(g'(x,[0])) \to 0 \quad g'(g'(x,z_2),z_1) \to g'(x,[\,])\,.$$

Now, starting from the transformed initial term $t' = f(g'(g'(0,[\,]),[\,]))$ which corresponds to t above, there is only one normal form as desired, namely 0.

References

1. Serbanuta, T.F., Rosu, G.: Computationally equivalent elimination of conditions. *Proc. 17th RTA*, LNCS 4098, Springer (2006) 19–34
2. Gmeiner, K.: Transformations of conditional term rewriting systems. Master's thesis, TU Wien (2007)
3. Bergstra, J., Klop, J.: Conditional rewrite rules: Confluence and termination. Journal of Computer and System Sciences **32**(3) (1986) 323–362
4. Giovanetti, E., Moiso, C.: Notes on the elimination of conditions. *Proc. 1st CTRS, Orsay, France, 2007*, LNCS 308, Springer (1988) 91–97
5. Viry, P.: Elimination of conditions. J. Symb. Comput. **28**(3) (1999) 381–401
6. Marchiori, M.: Unravelings and ultra-properties. In Hanus, M., Rodríguez-Artalejo, M.M., eds., Proc. 5th ALP, LNCS 1139, Springer (September 1996) 107–121
7. Ohlebusch, E.: Advanced Topics in Term Rewriting. Springer (2002)
8. Antoy, S., Brassel, B., Hanus, M.: Conditional narrowing without conditions. In: *Proc. 5th PPDP*, ACM Press (2003) 20–31
9. Rosu, G.: From conditional to unconditional rewriting. In: *Proc. 17th WADT*, LNCS 3423, Springer (2004) 218–233
10. Nishida, N., Mizutani, T., Sakai, M.: Transformation for refining unraveled conditional term rewriting systems. *ENTCS* 174(10), 2007.
11. Schernhammer, F., Gramlich, B.: On proving and characterizing operational termination of deterministic conditional rewrite systems. In: *Proc. 9th WST*, Paris, France. (2007) 82–85
12. Lucas, S., Meseguer, J., Marché, C., Urbain, X.: Proving operational termination of membership equational programs. Higher-Order and Symbolic Computation **21**(1–2) (1998) 59–88

A declarative debugger for Maude ⋆ (Extended Abstract)

A. Riesco, A. Verdejo, R. Caballero, and N. Martí-Oliet

Facultad de Informática, Universidad Complutense de Madrid, Spain
ariesco@fdi.ucm.es, alberto@sip.ucm.es, rafa@sip.ucm.es, narciso@sip.ucm.es

Declarative debugging, introduced by E. Y. Shapiro [1], is a semi-automatic technique that starts from a computation considered incorrect by the user (error symptom) and locates a program fragment responsible for the error. It has been widely employed in the logic [2], functional [3], and multiparadigm [4] programming languages. A *debugging tree* is used as a logical representation of the computation; each node represents the result of a computation step, which must follow from the results of its child nodes by some logical inference. Diagnosis proceeds by traversing the debugging tree, asking questions to an external oracle (generally the user) until a so-called *buggy node* is found. Any buggy node represents an erroneous computation step, and the debugger can display the program fragment responsible for it.

Maude [5] is a declarative language based on both equational and rewriting logic for the specification and implementation of a whole range of systems. Specifications in Maude are called modules, and can be either *functional* or *system* modules. Functional modules define data types and operations on them by means of *membership equational logic* theories that support multiple sorts, subsort relations, equations, and assertions of membership in a sort. Declarative debugging of functional modules has been presented in [6, 7], and exploits the fact that these modules are expected to be terminating, confluent, and sort decreasing. System modules specify rewrite theories that also support *rules*, defining local concurrent transitions that can take place in a system. Since the rules in system modules are not assumed to be either confluent or terminating, they require a new treatment in the debugging process, different to the one developed for functional modules and in general to that of functional languages. In our current work, we have been able to develop a declarative debugger that integrates both functional and system modules, treating appropriately equations, memberships, and rewrite rules.

The debugging process starts with an incorrect computation (a reduction, a type inference, or a rewrite) from an initial term. Our debugger, after building a proof tree for that inference, will present to the user questions about the computation. Moreover, since the questions are located in the proof tree, the answer allows the debugger to discard a subset of the questions, leading and shortening the debugging process. The proof tree is built by using the inference rules of rewriting logic. However, we do not use directly this proof tree as debugging tree, but a suitable abbreviation, which *reduces and simplifies* the questions that will be asked to the user while keeping the soundness and completeness of the technique. In particular this abbreviated tree contains only

⋆ Research supported by MEC Spanish projects *DESAFIOS* (TIN2006-15660-C02-01) and *MERIT-FORMS* (TIN2005-09027-C03-03), and Comunidad de Madrid program *PROMESAS* (S-0505/TIC/0407).

nodes corresponding to inference rules that apply statements included in the specification, which are the only possible buggy nodes. Nodes related with the congruence or transitivity inference rules, for example, are clearly correct.

In the case of functional modules, the debugger builds a debugging tree whose nodes give rise to questions of the form "Is it correct that T fully reduces to T'?" which in general are easier to answer. However, in the absence of confluence and termination, this kind of questions does not make sense; thus, in the case of system modules, we have decided to develop two different trees whose nodes produce questions of the form "Is it correct that T is rewritten to T'?" where the difference consists in the number of steps involved in the rewrite. While one of the trees refers only to one-step rewrites, which are often easier to answer, the other one can also refer to many-steps rewrites that, although may be harder to answer, in general allow to discard a bigger subset of nodes. The user, depending on the debugged specification or his "ability" to answer questions involving several rewrite steps, can choose between these two kinds of trees.

The current version of the tool has the following characteristics:

- It supports all kinds of modules: for example, operators can be declared with any combination of axiom attributes (except for the attribute `strat`, that allows to specify an evaluation strategy); equations can be defined with the `otherwise` attribute; modules can be parameterized; and operators' arguments can be `frozen` (see [5] for the meaning of all these concepts).
- It allows to debug specifications where some statements are suspicious and have been labeled (each one with a different label). Only these labeled statements generate nodes in the proof tree, while the unlabeled ones are considered correct. The user is in charge of this labeling.
- The user can decide to use all the labeled statements as suspicious or can use only a subset by trusting labels and modules. Moreover, the user can answer that he trusts the statement associated with the currently questioned inference; that is, statements can be trusted "on the fly." This produces that other nodes associated with the currently trusted statement are also deleted from the tree.
- In case of debugging a rewrite computation, two debugging trees can be built: one whose questions are related to one-step rewrites and another whose questions are related to several steps. The latter tree is partially built so that any node corresponding to a one-step rewrite is expanded only when the navigation process reaches it.
- It provides two strategies to traverse the tree: the more intuitive *top-down* strategy, that traverses the tree from the root asking each time for the correctness of all the children of the current node, and then continues with one of the incorrect children; and the more efficient *divide and query* strategy, that each time selects the node whose subtree's size is the closest one to half the size of the whole tree, keeping only this subtree if its root is incorrect, and deleting the whole subtree otherwise.
- The user can answer "don't know," which avoids the question by asking alternative ones. Since the information provided by the user in this case is incomplete, it is not assured to find the wrong statement.
- Before starting the debugging process, the user can select a module containing only correct statements. By checking the correctness of the inferences with respect to this module (i.e., using this module as oracle) the debugger can reduce the number

of questions asked to the user. Notice that the information provided by this module need not be complete, in the sense that some functions can be only partially defined.
– It provides an `undo` command, that allows the user to return to the previous state when a wrong answer has been provided.

The Maude system includes the predefined `META-LEVEL` module supporting reflection in rewriting logic [5, Chap. 14]. This powerful feature allows access to metalevel entities such as specifications or computations as usual data. Therefore, we are able to generate and navigate the debugging tree of a Maude computation using operations in Maude itself. In addition, the Maude system provides another module, `LOOP-MODE` [5, Chap. 17], which can be used to specify input/output interactions with the user. However, instead of using this module directly, we extend Full Maude [5, Chap. 18], that includes features for parsing, evaluating, and pretty-printing terms, improving the input/output interaction. Moreover, Full Maude allows the specification of concurrent object-oriented systems and parameterized ones, that can also be debugged. Thus, our declarative debugger, including its user interactions, is implemented in Maude itself. Complete explanations about the fundamentals and novelties of our debugging approach can be found in [8], which, together with the source files for the debugger, examples, and related papers, is available from `maude.sip.ucm.es/debugging`

We plan to improve the interaction with the user by providing a complementary graphical interface that allows the user to navigate the tree with more freedom. We are also studying how to handle the `strat` operator attribute, that allows the specifier to define an evaluation strategy. This can be used to represent some kind of laziness. Other future work will include how to debug *missing answers* [9] in addition to the wrong answers we have treated thus far.

References

1. Shapiro, E.: Algorithmic Program Debugging. ACM Dist. Dissertation. MIT Press (1983)
2. Lloyd, J.W.: Declarative error diagnosis. New Generation Computing **5**(2) (1987) 133–154
3. Nilsson, H., Fritzson, P.: Algorithmic debugging of lazy functional languages. Journal of Functional Programming **4**(3) (1994) 337–370
4. Caballero, R., Rodríguez-Artalejo, M.: DDT: A declarative debugging tool for functional-logic languages. In: Proc. 7th International Symposium on Functional and Logic Programming (FLOPS'04). Volume 2998 of Lect. Notes in Comp. Sci., Springer (2004) 70–84
5. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude: A High-Performance Logical Framework. Volume 4350 of Lect. Notes in Comp. Sci. Springer (2007)
6. Caballero, R., Martí-Oliet, N., Riesco, A., Verdejo, A.: Declarative debugging of membership equational logic specifications. In Degano, P., Nicola, R.D., Meseguer, J., eds.: Concurrency, Graphs and Models. Volume 5065 of Lect. Notes in Comp. Sci., Springer (2008) 174–193
7. Caballero, R., Martí-Oliet, N., Riesco, A., Verdejo, A.: A declarative debugger for Maude functional modules. In: Proceedings Seventh International Workshop on Rewriting Logic and its Applications, WRLA 2008, Elsevier (2008) To appear.
8. Riesco, A., Verdejo, A., Caballero, R., Martí-Oliet, N.: Declarative debugging of Maude modules. Technical Report SIC-6/08, Dpto. Sistemas Informáticos y Computación, Universidad Complutense de Madrid (2008) `http://maude.sip.ucm.es/debugging`.
9. Naish, L.: Declarative diagnosis of missing answers. New Generation Computing **10**(3) (1992) 255–286

A Rewrite Approach for Pattern Containment

Barbara Fila–Kordy
barbara.kordy@univ-orleans.fr

LIFO - Université d'Orléans, France

1 Introduction

Every XML document is generally represented as a tree, and XPath is the main language for navigating and selecting nodes in XML documents [1]. The focus in this work is on the containment problem [2, 3] for the fragment XP(/,//,[ ],$*$) of XPath. Any element of XP(/,//,[ ],$*$) is a query that can be represented as a rooted tree structure graph, called *pattern*, which can have edges of two types: *child* and *descendant*. A given XML tree t is said to be a *model* of a given pattern P iff there exists an *embedding* from P to t, defined as a root–, symbol– and path–preserving function, from $Nodes(P)$ to $Nodes(t)$, cf. [2]. A pattern P is *contained* in a pattern Q ($P \subseteq Q$) iff any model of P is also a model of Q. Miklau and Suciu prove in [2] that the containment problem is CoNP–complete. They also give a sufficient (but not necessary) condition for pattern containment. We propose to handle the pattern containment problem using a rewrite approach. We define a set $\mathcal{R}$ of rewrite rules based on the semantics of XP(/,//,[ ],$*$)–query containment, and show that $P \subseteq Q$ if and only if we can rewrite P to Q, using these rules.

2 The Approach

Patterns can be formally defined as the expressions P derived form the following grammar, where $\omega \in \Sigma \cup \{*\}$ (Σ is a given alphabet, and '$*$' is the *don't–care* symbol of XPath):

$$\begin{array}{lll} M: & \varepsilon \mid \downarrow \omega \mid \Downarrow \omega \mid MM & // \text{ path} \\ S: & \emptyset \mid \{MS\} \mid S \cup S & // \text{ set of sibling unrooted terms} \\ P: & \omega MS & // \text{ patterns} \end{array}$$

We call a *term* any expression of the type M, S or P derived from this grammar, as well as any finite disjunction $P_1 \vee P_2 \vee \cdots \vee P_n$ of the patterns. The terms in M and S are *unrooted*, those in P are *rooted*. Given patterns P and P_i, for $1 \leq i \leq n$, the terms of the form ε, P or $P_1 \vee \cdots \vee P_n$, will be called *d–patterns*. A disjunctive d–pattern represents different models of a given pattern. We present below our set $\mathcal{R}$ of rewrite rules for rewriting rooted and unrooted terms. Let M, S, P (possibly with primes, subscripts) be as in the grammar above, D stands for a d–pattern, $\sigma \in \Sigma$, and $\omega, \omega' \in \Sigma \cup \{*\}$:

1. $S \longrightarrow \emptyset$, $M \longrightarrow \varepsilon$ //cut;

2. $M\sigma S \longrightarrow M * S$, $M\sigma M' \longrightarrow M * M'$ //replace any symbol of Σ by the '$*$' of XPath;
3. $\downarrow \omega S \longrightarrow \Downarrow \omega S$ //every child is also a descendant;
4. $\xi\omega\xi'\omega' S \longrightarrow \Downarrow \omega' S$, where $\xi, \xi' \in \{\downarrow, \Downarrow\}$ //ignore an intermediate node
5. $\{M\{S_1, S_2\}\} \longrightarrow \{MS_1, MS_2\}$ //left distributivity;
6. $S \longrightarrow S \cup S'$, where $S \longrightarrow S'$ //add new siblings;
7. $S \cup S_1 \longrightarrow S' \cup S_1$, if $S \longrightarrow S'$ //rewrite some of the siblings;
8. $\Downarrow \omega S \longrightarrow (\downarrow \omega S) \vee (\downarrow * \Downarrow \omega S)$ //case analysis: descendant is either a child or has depth ≥ 2,
9. $\Downarrow \omega S \longrightarrow (\downarrow \omega S) \vee (\Downarrow * \downarrow \omega S)$ //idem;
10. $D \vee P \longrightarrow D \vee P'$, if $P \longrightarrow P'$ //case rewriting;
11. $P \vee P \vee D \longrightarrow P \vee D$ //consider any given case only once.

By *context–pattern* we mean any pattern having a special additional *hole* symbol $\langle\rangle$ that replaces one of its unrooted sub–terms; e.g. $\mathcal{C} = f\{\downarrow a, \Downarrow b\{\langle\rangle, \downarrow d\}, \Downarrow *\}$ is a context–pattern. To rewrite terms with the rules of $\mathcal{R}$ we use *suffix rewriting*: let X, X' be the unrooted terms, if $X \longrightarrow X' \in \mathcal{R}$, then for any context–pattern $\mathcal{C}$, we have $\mathcal{C}\langle\rangle X \longrightarrow \mathcal{C}\langle\rangle X'$, where $\mathcal{C}\langle\rangle X$ stands for the pattern obtained form $\mathcal{C}$ by replacing the hole symbol by the unrooted term X; e.g. for the context $\mathcal{C}$ given above, and the unrooted term $X = \Downarrow x\{\downarrow y, \Downarrow z\}$, we get the term $\mathcal{C}\langle\rangle X = f\{\downarrow a, \Downarrow b\{\Downarrow x\{\downarrow y, \Downarrow z\}, \downarrow d\}, \Downarrow *\}$. We also suppose that for any context–pattern $\mathcal{C}$ and any unrotted terms X et X', $\mathcal{C}\langle\rangle(X \vee X')$ is an abbreviated notation for the disjunctive d–pattern $(\mathcal{C}\langle\rangle X) \vee (\mathcal{C}\langle\rangle X')$.

The main result of our work is the following:

Theorem 1. *For any two patterns P and Q, $P \subseteq Q$ iff $P \xrightarrow{*}_{\mathcal{R}} Q$.*

The semantics of the rules guarantee that $P \xrightarrow{*}_{\mathcal{R}} Q$ implies $P \subseteq Q$. To show the converse, we first define a *morphism* from pattern P to pattern Q as a root–, symbol– and path–preserving function, from $Nodes(P)$ to $Nodes(Q)$. The authors of [2] prove that if there exists a morphism form a pattern Q to a pattern P, then $P \subseteq Q$, but the converse is not true. Figure 1 presents two patterns: $P = a \Downarrow b\{\downarrow c \downarrow * \Downarrow d, \downarrow b\{\downarrow c \Downarrow d, \downarrow b \downarrow c \downarrow d\}\}$, and $Q = a \Downarrow b\{\downarrow c \downarrow * \Downarrow d, \downarrow b \downarrow c \downarrow d\}$, such that $P \subseteq Q$, but there is no morphism from Q to P.

By a morphism from a pattern Q to a d–pattern of the form $P_1 \vee \cdots \vee P_n$, we mean a function which is a morphism from Q to P_i, for every $1 \leq i \leq n$. To prove the Theorem 1 we use the following results:

Lemma 1. *For any given patterns P and Q, if there exists a morphism from Q to P, then $P \xrightarrow{*}_{\mathcal{R}} Q$.*

Corollary 1. *For any given pattern Q and d-pattern D, if there exists a morphism from Q to D then $D \xrightarrow{*}_{\mathcal{R}} Q$.*

Proposition 1. *Given two patterns P and Q. If $P \subseteq Q$, then one can construct a d–pattern D verifying $P \xrightarrow{*}_{\mathcal{R}} D$, s.t. there exists a morphism from Q to D.*

Example 1. Let us consider the patterns P and Q from the Figure 1. Below we show how to rewrite P into Q and thus how to prove the containment $P \subseteq Q$:

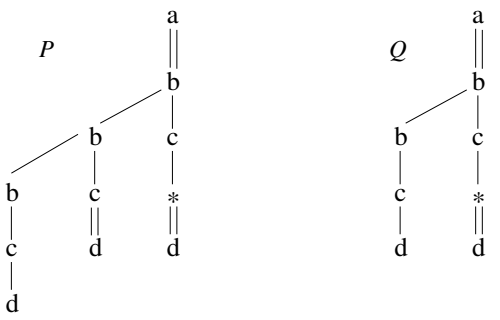


Fig. 1. Patterns P and Q, such that $P \subseteq Q$, but no morphism from Q to P

$$P = a \Downarrow b\{\downarrow c \downarrow * \Downarrow d, \downarrow b\{\downarrow c \Downarrow d, \downarrow b \downarrow c \downarrow d\}\} \xrightarrow{8}$$
$$a \Downarrow b\{\downarrow c \downarrow * \Downarrow d, \downarrow b\{\downarrow c \downarrow d, \downarrow b \downarrow c \downarrow d\}\} \vee$$
$$a \Downarrow b\{\downarrow c \downarrow * \Downarrow d, \downarrow b\{\downarrow c \downarrow * \Downarrow d, \downarrow b \downarrow c \downarrow d\}\} \xrightarrow{1}$$
$$a \Downarrow b\{\downarrow c \downarrow * \Downarrow d, \downarrow b\{\downarrow c \downarrow d\}\} \vee a \Downarrow b \downarrow b\{\downarrow c \downarrow * \Downarrow d, \downarrow b \downarrow c \downarrow d\}\} \xrightarrow{4}$$
$$a \Downarrow b\{\downarrow c \downarrow * \Downarrow d, \downarrow b \downarrow c \downarrow d\} \vee a \Downarrow b\{\downarrow c \downarrow * \Downarrow d, \downarrow b \downarrow c \downarrow d\} \xrightarrow{11}$$
$$a \Downarrow b\{\downarrow c \downarrow * \Downarrow d, \downarrow b \downarrow c \downarrow d\} = Q.$$

Further Remarks

- Our rewrite system is non–deterministic, nevertheless if P and Q are given, there exists a well–defined, goal–directed strategy permitting to rewrite P into Q (as in Example 1).
- Our approach is no longer valid, if the terms are not rewritten using suffix rewriting; e.g. $P = (* \Downarrow *) \subseteq Q = (* \downarrow *)$, but $P \downarrow a = (* \Downarrow * \downarrow a)$ is *not* contained in $Q \downarrow a = (* \downarrow * \downarrow a)$.

All the results obtained in this work remain valid even if the models of patterns are given in a compressed form (e.g. DAGs instead of trees, as in [4]). The rewrite approach that we have presented here can be adapted for evaluating unary or n–ary queries, by using suitably defined *marked patterns*. We hope to extend our results to more general patterns, having both descendant and ascendant edges.

References

1. World Wide Web Consortium: XML Path Language. Available on: `http://www.w3.org/TR/xpath` (1999) W3C Recommendation 16 November 1999.
2. Miklau, G., Suciu, D.: Containment and Equivalence for a Fragment of XPath. J. ACM **51**(1) (2004) 2–45
3. Neven, F., Schwentick, T.: XPath Containment in the Presence of Disjunction, DTDs, and Variables. In: ICDT '03: Proceedings of the 9th International Conference on Database Theory, London, UK, Springer-Verlag (2002) 315–329
4. Fila, B., Anantharaman, S.: Automata for Positive Core XPath Queries on Compressed Documents. In: Proceedings of LPAR'06, Springer-Verlag (2006) 467–481

Symbolic semantics for cc-pi: an algebraic view

Filippo Bonchi[1], Maria Grazia Buscemi[2], and Ugo Montanari[1]

[1] Dipartimento di Informatica, Università di Pisa,
[2] IMT Lucca Institute for Advanced Studies
fibonchi@di.unipi.it, m.buscemi@imtlucca.it, ugo@di.unipi.it

The operational semantics of process calculi is usually specified by a labeled transition systems (LTS). The abstract semantics is given in terms of behavioural equivalence, often bisimilarity. A key property is that behavioural equivalence be a congruence, i.e. that abstract semantics is compositional with respect to the algebraic operations of the language. In order to obtain a congruence that is also a bisimilarity, one can consider the largest bisimulation that is closed under all contexts, in short, the *largest bisimulation congruence*. An equivalent approach is to introduce additional moves of the form $p \xrightarrow{c,a} q$, for every context c, whenever $c[p] \xrightarrow{a} q$ is a transition in the original LTS. If we call *saturated* the resulting LTS, we have that ordinary bisimilarity on the saturated LTS coincides with the largest bisimulation congruence (on the original LTS). By analogy we call the latter *saturated bisimilarity*.

This idea was originally introduced by Sassone and the third author in [1]. They define *dynamic bisimilarity* in order to make weak bisimilarity of CCS [2] a congruence. Analogously, since late and early bisimilarity of π-calculus [3] are not preserved under substitution (and thus under input prefixes), in [4] Sangiorgi introduces *open bisimilarity* ($\sim^o$) as the largest bisimulation on π-calculus agents which is closed under substitutions. Another example of saturated bisimilarity is $\sim^1$ [5] for the asynchronous π-calculus [5, 6]. Here the basic bisimilarity, namely $o\tau$-bisimilarity, is not a congruence under parallel outputs, and thus at any step of definition of $\sim^1$ the observer inserts the process in parallel with all possible output messages.

Saturated bisimilarity is quite hard to check as the portion of LTS reachable in a step by any nontrivial agent is usually infinite. Sangiorgi defines in [4] a symbolic notion of bisimilarity for the π-calculus and proves that it coincides with $\sim^O$. Analogously in [5], Amadio et al. define a so-called asynchronous bisimilarity that coincides with $\sim^1$. The rough idea behind symbolic bisimulation [7] is to define a transition system whose transitions are labelled by the minimal contexts that allow the transitions to take place.

Inspired by Leifer and Milner's reactive system [8], the first and the third author have introduced in [9] the theory of *context interactive system*: an algebraic framework that allows to systematically define symbolic bisimilarity in such a way that it coincides with the saturated one. This construction employs some general knowledge about the modeled formalism. For instance, in π-calculus (without mismatch) it is possible to prove [3] that:

$$\forall \text{ process } p \text{ and substitution } \sigma, \text{ if } p \xrightarrow{\mu} q \text{ then } \sigma(p) \xrightarrow{\sigma(\mu)} \sigma(q).$$

Thus, if in the saturated LTS, $p \xrightarrow{\phi,\mu} p'$ (meaning that $\phi(p) \xrightarrow{\mu} p'$), then surely also $p \xrightarrow{\psi(\phi),\psi(\mu)} \psi(p')$. The second transition is to some extent *redundant*, i.e., we can ig-

nore it without changing the saturated bisimilarity. For any formalism, we identify a set of rules T (given in a fixed format) expressing how contexts modify transitions, and we define an inference relation $\vdash_T$ amongst the transitions of the saturated transition system. At this point, the symbolic transition system is the saturated transition system without redundant transitions, i.e. the minimal LTS β such that:

$$p \xrightarrow{c,o} q \text{ if and only if } p \xrightarrow{c',o'}_{\beta} q' \text{ and } p \xrightarrow{c',o'} q' \vdash_T p \xrightarrow{c,o} q.$$

Unfortunately, the standard notion of bisimilarity over β yields an equivalence that is usually stricter than saturated bisimilarity. Thus, instead of requiring the syntactic correspondence of labels in the bisimulation game, we require that:

$$\text{if } p \xrightarrow{c,o}_{\beta} p_1, \text{ then } q \xrightarrow{c',o'}_{\beta} q_1' \text{ such that } q \xrightarrow{c',o'} q_1' \vdash_T q \xrightarrow{c,o} q_1 \text{ and } p_1 R q_1.$$

The main theorem in [9] guarantees that symbolic bisimilarity (i.e., the largest bisimulation defined as above) coincides with saturated bisimilarity. In [9] the authors show that this theorem exactly subsumes that of [4] and [5]. In this work we show that the above framework also applies to the cc-pi calculus.

The cc-pi calculus [10] combines two main programming paradigms: name-passing calculi (see e.g. [3, 11]) and concurrent constraint programming [12]. On the one side, cc-pi inherits from the explicit fusion calculus [11] a symmetric, synchronous mechanism of interaction between senders and receivers, where the sent name is 'fused' (i.e., identified) to the received name and such *explicit fusion* allows to use interchangeably the two names. On the other side, cc-pi generalises explicit fusions to be arbitrary constraints and introduces primitives for creating and making logical checks on constraints. Specifically, a cc-pi process $p = c \,|\, \texttt{tell}\ c'.q$ can place a constraint c' and then evolve to the parallel composition of $c \times c'$ and q, if the combination of constraints $c \times c'$ is *consistent*. Similarly, $p = c \,|\, \texttt{ask}\ c'.q$ makes a transition to q if the constraint c' is *entailed* by c. Moreover, a process $(x = v) \times (v = y) \,|\, \overline{x}\langle z\rangle.p' \,|\, y\langle w\rangle.q'$, with $\overline{x}\langle z\rangle$ an output action and $y\langle w\rangle$ an input action, can make a synchronisation because the identification of the names x and y is entailed by the constraints in parallel. Such an interaction yields the fusion $z = w$ that is consistent with the other constraints:

$$(x = v) \times (v = y) \,|\, \overline{x}\langle z\rangle.p' \,|\, y\langle w\rangle.q' \xrightarrow{\tau} (x = v) \times (v = y) \times (z = w) \,|\, p' \,|\, q'.$$

This transition corresponds to the simultaneous execution of $\texttt{ask}\ x = y$ and $\texttt{tell}\ z = w$.

In [13] the second and the third authors define a saturated bisimilarity for cc-pi. The basic idea is that constraints running in parallel with a process have an effect on the names of that process as they can allow or disallow transitions. Hence, the natural adaptation of saturated bisimilarity to cc-pi is to replace contexts with constraints in parallel. For instance, the process $\overline{x}\langle z\rangle.\mathbf{0} \,|\, y\langle w\rangle.\mathbf{0}$ that tries to synchronize on channels with different names and the inert process $\mathbf{0}$ are not bisimilar since, in the context $x = y \,|\, \_$, the first one can make a move while the second one is stuck.

In [13] the authors also present a symbolic bisimulation for cc-pi and they show that the two given notions coincide. The symbolic LTS features labels representing the 'least restrictive' constraints that enable process moves. Such labels exploit the division

$\div$ operator over c-semirings [14]. As an example, consider the transitions below:

$$c \,|\, \mathtt{ask}\ d.p \xrightarrow{d \div c, \tau} c \times (d \div c) \,|\, p$$
$$c \,|\, \overline{x}\langle z\rangle.p' \,|\, y\langle w\rangle.q' \xrightarrow{(x=y) \div c, \tau} c \times ((x=y) \div c) \times (z=w) \,|\, p' \,|\, q'$$

with $d \div c$ the weakest constraint such that $c \times c'$ entails d, and similarly for $(x=y) \div c$.

The main contribution of the present work is to show that the symbolic semantics of cc-pi [13] is an instance of the general framework proposed in [9]. This result shows, from a side, the generality of [9], from the other, the canonicity of [13], since that approach is analogous to [5, 4, 11]. Moreover, our algebraic framework provides a final semantics through *normalized coalgebras* [15], thus ensuring the existence of a minimal representative and a minimization procedure to check bisimilarity.

References

1. Montanari, U., Sassone, V.: Dynamic congruence vs. progressing bisimulation for CCS. Fundamenta Informaticae **16**(1) (1992) 171–199
2. Milner, R.: Communicating and Mobile Systems: the π-Calculus. Cambridge University Press (1999)
3. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, i and ii. Information and Computation **100**(1) (1992) 1–40, 41–77
4. Sangiorgi, D.: A theory of bisimulation for the pi-calculus. Acta Informatica **33**(1) (1996) 69–97
5. Amadio, R.M., Castellani, I., Sangiorgi, D.: On bisimulations for the asynchronous pi-calculus. In: Proc. of CONCUR '96. Volume 1119 of LNCS., Springer (1996) 147–162
6. Honda, K., Tokoro, M.: An object calculus for asynchronous communication. In: Proc. of ECOOP '91. Volume 512 of LNCS., Springer (1991) 133–147
7. Hennessy, M., Lin, H.: Symbolic bisimulations. Theoretical Computer Science **138**(2) (1995) 353–389
8. Leifer, J.J., Milner, R.: Deriving bisimulation congruences for reactive systems. In: Proc. of CONCUR '00. Volume 1877 of LNCS., Springer (2000) 243–258
9. Bonchi, F., Montanari, U.: Symbolic semantics revisited. In: Proc. of FOSSACS '08. Volume 4962 of LNCS., Springer (2008) 395–412
10. Buscemi, M.G., Montanari, U.: Cc-pi: A constraint-based language for specifying service level agreements. In: Proc. of ESOP '07. Volume 4421 of LNCS., Springer (2007) 18–32
11. Wischik, L., Gardner, P.: Explicit fusions. Theoretical Computer Science **340(3)** (2005) 606–630
12. Saraswat, V., Rinard, M.: Concurrent constraint programming. In: Proc. of POPL '90, ACM Press (1990)
13. Buscemi, M.G., Montanari, U.: Open bisimulation for the concurrent constraint pi-calculus. In: Proc. of ESOP '08. Volume 4960 of LNCS., Springer (2008) 254–268
14. Bistarelli, S., Gadducci, F.: Enhancing constraints manipulation in semiring-based formalisms. In: Proc. of ECAI '06, IOS Press (2006) 63–67
15. Bonchi, F., Montanari, U.: Coalgebraic models for reactive systems. In: Proc. of CONCUR '07. Volume 4701 of LNCS., Springer (2007) 364–380

A coalgebraic characterization of behaviours in the linear time – branching time spectrum

Luís Monteiro

CITI, Departamento de Informática,
Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, Portugal
lm@di.fct.unl.pt

1 Introduction

When coalgebras are viewed as models of general dynamical systems [1], final coalgebras appear as abstract descriptions of the observable behaviours of such systems. For many systems, however, several notions of behaviour have been proposed, depending on the point of view or on the intended application. A case in point is the class of (labeled) transition systems: in [2], twelve notions of behaviour are presented and hierarchically organized in the so-called linear time – branching time spectrum. The problem is then how to characterize in coalgebraic terms different notions of behaviour for a given type of systems. A characterization of traces for some categories of coalgebras appear in [3–5]. Here we shall confine ourselves to a category of transition systems and characterize coalgebraically a representative set of behaviours from the linear time – branching time spectrum, namely, traces, ready-traces and failures; we believe most other kinds of behaviours in the spectrum can be treated in a similar way.

2 The General Framework

Our approach differs from the one in the cited papers, which is based on monads and distributive laws. We start from a category $\mathbf{C}$ (of transition systems) equipped with a faithful functor $U : \mathbf{C} \to \mathbf{Sets}$ where we assume $Uf = f$ for arrows f. For each type of behaviours (traces, ready-traces or failures) we define a full subcategory $\mathbf{D}$ whose behaviours are precisely those we wish to capture. More precisely, the given behaviours can be structured as a final object Z in $\mathbf{D}$. To associate behaviours of the given kind with a transition system S we introduce a functor T from $\mathbf{C}$ to $\mathbf{C}$ whose image is contained in $\mathbf{D}$. There is a unique morphism $\beta : T\mathsf{S} \to \mathsf{Z}$, so the last step is to define $\eta_{\mathsf{S}} : U\mathsf{S} \to UT\mathsf{S}$ in order to associate the intended behaviours with the given transition system by the "behaviour map" $\mathsf{beh}_{\mathsf{S}} = \beta \circ \eta_{\mathsf{S}}$. Ideally, we would like η to be a natural transformation $\mathrm{Id}_{\mathbf{C}} \to T$, but as it turns out η_{S} is not in general a morphism in $\mathbf{C}$. Thus, η is just a natural transformation $U \to UT$. We may assume, however, that η_{S} is a morphism if S is in $\mathbf{D}$.

It will be useful to characterize Z as a final object in a category $\mathbf{C}_T$ with the same objects as $\mathbf{C}$ but with more morphisms. Say a function $f : U\mathsf{S}_1 \to$

$U\mathsf{S}_2$ is a "T-morphism" if there exists a morphism $f' : T\mathsf{S}_1 \to T\mathsf{S}_2$ such that $f' \circ \eta_{\mathsf{S}_1} = \eta_{\mathsf{S}_2} \circ f$. Morphisms f and the functions η_{S} are T-morphisms: just take respectively $f' = T(f)$ and $\eta'_{\mathsf{S}} = \eta_{T\mathsf{S}}$. The morphisms of $\mathbf{C}_T$ are, by definition, the T-morphisms. The next result shows that Z is final in $\mathbf{C}_T$, so $\mathsf{beh}_{\mathsf{S}} : \mathsf{S} \to \mathsf{Z}$ is the unique T-morphism.

Theorem 1. *An object* Z *of* $\mathbf{D}$ *is final in* $\mathbf{D}$ *iff* Z *is final in* $\mathbf{C}_T$ *and the unique* T*-morphism* $T\mathsf{Z} \to \mathsf{Z}$ *is a morphism.*

To compare different notions of behaviour, suppose $\mathbf{C}$ also has a final object.

Corollary 1. *Suppose* $\mathbf{C}$ *has a final object* W *and* $\mathbf{C}_T$ *has a final object* Z*. Then* Z *is a subobject of* W *in* $\mathbf{C}$ *and a retract of* W *in* $\mathbf{C}_T$*.*

"Behavioural equivalences" $\overset{\mathsf{Z}}{=}_{\mathsf{S}}$ and $\overset{\mathsf{W}}{=}_{\mathsf{S}}$ on $U\mathsf{S}$ are defined as usual as the kernels of $\mathsf{beh}^{\mathsf{Z}}_{\mathsf{S}} : \mathsf{S} \to \mathsf{Z}$ and $\mathsf{beh}^{\mathsf{W}}_{\mathsf{S}} : \mathsf{S} \to \mathsf{W}$ respectively.

Corollary 2. *The equivalence* $\overset{\mathsf{W}}{=}_{\mathsf{S}}$ *is finer than* $\overset{\mathsf{Z}}{=}_{\mathsf{S}}$*, that is,* $\overset{\mathsf{W}}{=}_{\mathsf{S}} \subseteq \overset{\mathsf{Z}}{=}_{\mathsf{S}}$*.*

3 Traces, Ready-traces, Failures

The preceding considerations hold in any category $\mathbf{C}$. Let us see how the introduced notions materialize in each of the three cases mentioned above.

Traces We fix a set A of "actions" and consider transition systems as coalgebras $\mathsf{S} = \langle S, \psi : S \to \mathcal{P}(S)^A \rangle$; this is our category $\mathbf{C}$. The forgetful functor U is given by $U\mathsf{S} = S$ and $Uf = f$. As usual we write $s \xrightarrow{a} s'$ if $s' \in \psi(s)(a)$ and extend this notation to $s \xrightarrow{x} s'$ for any word $x \in A^*$; such a word x is then called a "trace" of s (note that in [3–5] systems have a notion of "successful" termination and only so-called "complete" traces are considered, but for our purposes the notion presented here is simpler); the set of all traces of s is nonempty and prefix-closed. For $\mathbf{D}$ we take the category of all deterministic transitions systems, that is, such that each $\psi(s)(a)$ has cardinality at most one. The final system $\mathsf{Z} = \langle Z, \zeta \rangle$ has Z the set of all nonempty and prefix-closed languages over A and ζ defined as for arbitrary languages (see [1]). The functor T is the familiar powerset construction (see [1]) and $\eta_{\mathsf{S}} : S \to \mathcal{P}(S)$ for $\mathsf{S} = \langle S, \psi \rangle$ is the function $s \mapsto \{s\}$.

Ready-traces Given S and $s \in S$, let $I(s) = \{a \mid \psi(s)(a) \neq \emptyset\}$. The ready-traces of S are by definition the traces of $\langle S, \bar{\psi} : S \to \mathcal{P}(S)^{A \times \mathcal{P}(A)} \rangle$, where $\bar{\psi}(s)(a, X) = \{s' \in \psi(s)(a) \mid I(s') = X\}$; we abbreviate $(a_1, X_1) \cdots (a_n, X_n)$ to $a_1 X_1 \cdots a_n X_n$ (in the standard definition the ready-traces start with a set X_0 of actions, but this is unnecessary and inconvenient for our purposes). The category $\mathbf{D}$ is formed by the systems where $s_1, s_2 \in \psi(s)(a)$ and $s_1 \neq s_2$ imply $I(s_1) \neq I(s_2)$. The final system $\mathsf{Z} = \langle Z, \zeta \rangle$ has elements the nonempty and prefix-closed languages L over $A \times \mathcal{P}(A)$ such that $raX \in L$ and $b \in X$ imply $raXbY \in L$ for some Y; from the relation $L \xrightarrow{a,X} L'$, define $\zeta(L)(a) = \{\{L' \mid L \xrightarrow{a,X} L'\} \mid aX \in L\}$.

To define $T\mathsf{S}$, first let the set of "next initials" of $M \subseteq S$ after $a \in A$ be the set $NI(M,a) = \{I(s') \mid \exists s \in M, s \xrightarrow{a} s'\}$; then let $T\mathsf{S} = \langle \mathcal{P}(S), \tilde{\psi} \rangle$ with $\tilde{\psi}(M)(a) = \{\{s' \in \psi(s)(a) \mid s \in M, I(s') = X\} \mid X \in NI(M,a)\}$. The function $\eta_{\mathsf{S}} : S \to \mathcal{P}(S)$ is again $s \mapsto \{s\}$.

Failures Given a transition system as before, $(x, X) \in A^* \times \mathcal{P}(A)$ is a failure of s if $s \xrightarrow{x} s'$ and $X \cap I(s') = \emptyset$ for some s'. Let $C_s(x) = \{a \in A \mid xa \text{ is a trace of } s\}$. The objects in $\mathbf{D}$ are all systems such that: i) if $s \xrightarrow{a} s_1$, $s \xrightarrow{a} s_2$ and $s_1 \neq s_2$, then $I(s_1) \neq I(s_2)$; ii) if $s \xrightarrow{a} s'$ and $I(s') \subseteq J \subseteq C_s(a)$, then $s \xrightarrow{a} s''$ and $I(s'') = J$ for some s''; iii) if $s \xrightarrow{a} s_i \xrightarrow{b} s'_i$ $(i = 1, 2)$, then $s_i \xrightarrow{b} s'_{3-i}$ $(i = 1, 2)$. The final system $\mathsf{Z} = \langle Z, \zeta \rangle$ has Z the set of all "failure-sets" over A (subsets of $A^* \times \mathcal{P}(A)$ satisfying some axioms), called the "failures domain" in [2] (we omit the details). The functor T maps S to $\langle \hat{S}, \hat{\psi} \rangle$, where $\hat{S} = \{(M, X) \mid M \subseteq S, X \subseteq I(M) - I(s) \text{ for some } s \in M\}$ with $I(M) = \bigcup\{I(s) \mid s \in M\}$, and $\hat{\psi}$ is given by the transitions $(M, X) \xrightarrow{a} (M', X')$ for $a \in I(M) - X$ and $M' = \{s' \mid \exists s \in M, s \xrightarrow{a} s'\}$. The function $\eta_{\mathsf{S}} : S \to \hat{S}$ is given by $s \mapsto (\{s\}, \emptyset)$.

The inclusions $\mathbf{D}_{Tr} \to \mathbf{D}_{Fl} \to \mathbf{D}_{Rtr}$ reflect the relative positions of the corresponding behaviours in the spectrum and allow to relate the behaviours among themselves by the technique outlined above.

4 Concluding Remarks

It is our belief that the coalgebraic reconstruction of the behaviours in the linear time – branching time spectrum is an important contribution to our understanding of those behaviours. If in the process of doing so we enlarge our understanding of the notion of behaviour of a coalgebra (system), so much the better. Some ways to continue the work reported herein are to try to extend the outlined approach to the remaining cases in the spectrum and to relate our approach with the work in [3–5].

References

1. Rutten, J.: Universal coalgebra: a theory of systems. Theoretical Computer Science **249**(1) (2000) 3–80
2. van Glabbeek, R.: The linear time–branching time spectrum I: the semantics of concrete, sequential processes. In Bergstra, J., Ponse, A., Smolka, S., eds.: Handbook of process algebra, Elsevier (2001) 3–99
3. Power, J., Turi, D.: A coalgebraic foundation for linear time semantics. In Hofmann, M., Rosolini, G., Pavlovic, D., eds.: CTCS '99, Conference on Category Theory and Computer Science. Volume 29 of Electronic Notes in Theoretical Computer Science., Elsevier (1999) 259–274
4. Jacobs, B.: Trace semantics for coalgebras. In Adamek, J., Milius, S., eds.: Coalgebraic Methods in Computer Science. Volume 106 of Electronic Notes in Theoretical Computer Science., Elsevier (2004) 167–184
5. Hasuo, I., Jacobs, B., Sokolova, A.: Generic trace semantics via coinduction. Logical Methods in Computer Science **3**(4:11) (2007) 1–36

Parametric Contexts and Finitely Branching Bisimilarities for Process Calculi

Pietro Di Gianantonio Furio Honsell Marina Lenisa

Dipartimento di Matematica e Informatica, Università di Udine
{digianantonio,honsell,lenisa}@dimi.uniud.it

Abstract. We experiment Leifer-Milner *RPO approach* to CCS and to π-calculus. The basic category in which we carry out the construction is the category of term contexts. Several issues and problems emerge from this experiment; for them we propose some original solutions.

Recently, much attention has been devoted to derive *labelled transition systems* and *bisimilarity congruences* from *reactive systems*, in the context of process languages and graph rewriting, [1,2,3,4,5]. In the theory of process algebras, the operational semantics of CCS was originally given via a labelled transition system (LTS), while more recent process calculi have been presented via reactive systems plus structural rules. Reactive systems naturally induce behavioral equivalences which are congruences w.r.t. contexts, while LTS's naturally induce bisimilarity equivalences with coinductive characterizations. However, such equivalences are not congruences in general, or else it is an heavy, ad-hoc task to prove that they are congruences.

Leifer and Milner [1] presented a general categorical method, based on the notion of Relative Pushout (RPO), for deriving a transition system from a reactive system, in such a way that the induced *bisimilarity* is a *congruence*. The labels in Leifer-Milner's transition system are those contexts which are *minimal* for a given reaction to fire.

In the literature, some case studies have been carried out in the setting of process calculi, for testing the expressivity of Leifer-Milner's approach. Some difficulties have arisen in applying the approach directly to such languages, viewed as Lawvere theories, because of structural rules. Thus more complex categorical constructions have been introduced by Sassone and Sobocinski in [2].

In this work, we apply the RPO technique to the prototypical examples of CCS and pi-calculus.

Aims and basic choices are the following:
1. To consider simple and quite fundametal case studies in which to experiment the RPO approach.
2. To apply the RPO approach in the category of term contexts. In this category, arrows represent syntactic terms or contexts. The use of a category so strictly related to the original syntax has the advantage that the generated LTS has a quite direct and intuitive interpretation.

In carrying out the simpler case study given by CCS, we have found the following problems. For all of them we propose some original solutions.

– *Structural rules.* In [2], Sassone and Sobocinski proposed the use of G-categories to deal with reduction systems like CCS, where, beside the reduction rules, there is a series of structural rules. However, so far G-categories have been used to treat just tiny fragments of CCS, while in other treatments of CCS [4], structural rules are avoided through a graph encoding; namely there is a single graph representation for each class of structural equivalent terms. In this work, we show how, using a suitably defined G-category, one can directly apply the RPO approach to complete CCS calculus.
– *Names.* Another issue is given by names, and name biding. In this work we propose *de Bruijn indexes* as a suitable instrument to deal with the issues that name manipulation poses. We found out that de Bruijn indexes can be used also for π-calculus, where name manipulation is more sophisticated than in CCS.
– *Pruning the LTS.* The simple application of the RPO approach generates LTS's that are quite redundant, in the sense that most of the labels, and the corresponding transitions, can be eliminated from the LTS without affecting the induced bisimilarity. From a practical point of view, having such large trees makes the proofs of bisimilarity unnecessarily complex.

In this work, we propose a general technique that can be used in order to identify sets of labels that can be eliminated from the LTS, without modifying the induced bisimilarity. In detail, we introduce a notion of *definability* of a label in terms of a set of other labels C. We prove that, given a LTS L constructed via the RPO technique, if the a class C of labels is such that any other label in L is definable by C, then the restricted LTS L', obtained by considering only labels in C, induces the same bisimilarity of the original LTS L.

The result of the above technique is a LTS for CCS that coincides with the original LTS proposed by Milner.

Moreover, the above constructions and techniques can be applied, without any major problems, to the more sophisticated case given by the π-calculus. Also for the π-calculus we are able to derive a finitely branching LTS inducing a bisimilarity that is a congruence. Then so far the open question is to relate this LTS with the others presented in the literature.

References

1. Leifer, J.J., Milner, R.: Deriving bisimulation congruences for reactive systems. In: CONCUR. Volume 1877 of LNCS., Springer (2000) 243–258
2. Sassone, V., Sobocinski, P.: Deriving bisimulation congruences using 2-categories. Nord. J. Comput. **10**(2) (2003) 163–
3. Gadducci, F., Montanari, U.: Observing reductions in nominal calculi *ia* a graphical encoding of processes. In: Processes, Terms and Cycles. Volume 3838 of LNCS., Springer (2005) 106–126
4. Bonchi, F., Gadducci, F., König, B.: Process bisimulation *via* a graphical encoding. In: ICGT. Volume 4178 of LNCS., Springer (2006) 168–183
5. Bonchi, F., König, B., Montanari, U.: Saturated semantics for reactive systems. In: LICS, IEEE Computer Society (2006) 69–80

A Compositional Approach to Specification of Concurrent Systems*

Artur Zawłocki

Institute of Informatics, Warsaw University, Poland
zawlocki@mimuw.edu.pl

We present a mathematical semantics and a specification formalism for a software component framework. As a running example, let us consider a system consisting of three components: a Client communicating with a Server, and a mediating Channel which transports Client's requests and Server's responses.

We describe the behaviour of the system in terms of *events*, such as issuing a request or delivering a response. Additionally, events may carry data: integers, strings, session identifiers etc. Each component C is equipped with a set of events E_C with a distinguished subset $I_C \subseteq E_C$ of *internal* events. The intuition is that internal events are those over which the component has a complete control, while non-internal (or *external*) events require synchronisation with the component's environment. For instance, events of Client are of the form $inv(n)$, $repl(n)$ and $close$, where the parameter n represents an identifier used to correlate service invocations, represented by inv, with subsequent Server replies, represented by $repl$. Both inv and $close$ are classified as internal, which means that the Client decides when to submit a request (and with which identifier) or close the connection; on the other hand, the Client alone cannot force a reply to occur and hence $repl$ is an external event.

For describing component behaviour we propose an event-based temporal logic. Syntactically, it is the linear-time temporal logic LTL with past modalities extended with a construction for expressing "enabling" of events (which is a branching-time property). For instance, the formula

$$\forall n, n' \left(inv(n) \rightarrow \neg inv(n') \,\mathsf{U}\, (close \vee \Diamond repl(n)) \right)$$

states that after $inv(n)$ occurs no other inv event may occur until finally either $close$ occurs or $repl(n)$ is *enabled* (in other words, the Client is ready to execute $repl(n)$). Another property of Client is expressed by the formula

$$\forall n \left(\Diamond repl(n) \rightarrow \Diamond repl(n) \,\mathsf{W}\, (repl(n) \vee close) \right)$$

stating that once enabled, $repl(n)$ will be enabled as long as either $repl(n)$ or $close$ occurs.

The behaviour of Client is modelled by a set of sequences of events, called *runs*, representing complete executions of the component. As far as individual components are considered there is no difference between internal and external events. However, the distinction becomes important when we consider composition. In order to model composition we define when one component, C, is a

* This work was supported by the EU funded IST Project SENSORIA.

"subcomponent" of another one, D. One requirement is that events of C has to be included in those of D and, moreover, every internal event of C remains internal in D. Other requirements guarantee that runs of D are "simulated" by C and that internal events of C cannot be suppressed by D. In contrast, D may choose which of the external events of C that are internal in D should occur.

Using the notions of component containment and run restriction we can provide a semantics for an *architectural specification* of the overall system, similarly it as is done in CASL ([1]), from which the following syntax is borrowed:

$$\begin{aligned} &\textbf{archspec } \mathit{System} = \\ &\quad \textbf{units } \mathit{Chan} : \mathit{SP}_{\mathit{Chan}} \\ &\qquad\quad \mathit{Client} : \mathit{SP}_{\mathit{Client}} \textbf{ given } \mathit{Chan} \\ &\qquad\quad \mathit{Server} : \mathit{SP}_{\mathit{Server}} \textbf{ given } \mathit{Chan} \\ &\quad \textbf{result } \mathit{Client} \textbf{ and } \mathit{Server} \end{aligned}$$

Overall, our framework resembles categorical formalisms such as COMMUNITY ([2]), where components are synchronised by sharing common subcomponents (here, both Client and Server share the Channel), a complex system can be again viewed as a single component, and properties of the system can be inferred from the properties of its components. However, there are also some important differences. Most notably, we employ an expressive logic extending both LTL and the first-order logic and able to express enabling of events. Still, we are able to provide rather simple rules for inferring properties of complex systems. The main inference rule:

$$\frac{C \sqsubseteq D,\ C \models \phi}{D \models \phi} \quad \text{does not contain } \Diamond e \text{ for any } e \in (E_C \setminus I_C) \cap I_D$$

states that if C is a subcomponent of D and ϕ is a formula satisfied in C and not containing a subformula of the form $\Diamond e$ for any event e external for C but internal in D, then ϕ is satisfied in D as well. To obtain such a rule, we establish for our event-based logic an analogue of the *stuttering invariance* property of state-based logics such as the Lamport's TLA [3]. Unlike in TLA, this is achieved not by restricting the syntax of the logic but by adopting an appropriate, dense-time semantics (a similar approach was adopted in dense-time logic TLR, [4]). Finally, we consider a decidable propositional fragment of the logic, still expressive enough to describe many aspects of component interaction.

References

1. CoFI (The Common Framework Initiative), CASL *Reference Manual*, LNCS Vol. 2960 (IFIP Series), part III. Springer, 2004
2. J. L. Fiadeiro, A. Lopes and M. Wermelinger, *A Mathematical Semantics for Architectural Connectors*, LNCS 2793, pp. 190–234, Springer, 2003
3. L. Lamport, *The Temporal Logic of Actions*, ACM Toplas 16:3, pp. 872–923, 1994
4. H. Barringer, R. Kuiper and A. Pnueli, *A Really Abstract Concurrent Model and its Temporal Logic*, POPL '86: Proc. of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 173–183, ACM Press, 1986

Stone Duality for Nominal Sets

Vincenzo Ciancia, Fabio Gadducci

Department of Informatics, University of Pisa

1 Stating the problem

The syntax and semantics of *nominal calculi* has received great attention in computer science. Ordinary set-theoretical models proved inadequate to represent and reason about these languages, and new formalisms (mostly borrowed from category theory) had to be invented. On the side of the syntax, name *binding* is the most important and problematic construct. A categorial model that successfully addressed name binding was the presheaf category $\#\mathbf{1}^{\#\mathbf{1}}$ of sets indexed by finite sets and their morphisms. Independently, it was realized by Gabbay and Pitts that not all the arrows of $\#\mathbf{1}$ are needed to model binding, but *injective renamings* suffice. This was exploited in [1] to give a notion of *abstract syntax with variable binding*, using the category of *nominal sets*, i.e. sets equipped with a permutation action. These models have a long history: they were defined by Fraenkel and Mostowski to show independence of the axiom of choice from the other axioms of set theory. As it later turned on, this category is equivalent to the *Schanuel Topos*, to *permutation algebras* used to model the semantics of nominal calculi [2] and to *named sets* [3] (proof and explanation of many equivalence results in name-aware categories can be found in [4]).

Operations derived from the theory of nonimal sets were employed in [5] and subsequent works to provide models of a nominal spatial logic for concurrency, and in particular of operators that employ hidden names, such as the *freshness quantifier*. There, the notion of *Pset* (or *property set*) was introduced as a means to represent the semantics of the logic. Intuitively, a *Pset* is a set of processes closed under structural congruence and with respect to a suitable family of name substitutions. *Psets* do model the spatial logic in a neat way: however their usage was neither formally justified, nor put on a solid theoretical ground.

The work we present here stems from the observation that *Psets* are in turn a subalgebra of the power set (as observed by Caires and Cardelli), and in particular they are just obtained from the elements of the power set (viewed as a functor in the category of nominal sets) that are finitely supported. We thus view *Psets* as a subalgebra of the *ultrafilters* on the logics obtained by taking the tensor product of the boolean and permutation signature, that is, permutation-boolean algebras (notice that permutation algebras are just an algebraic definition of nominal sets). Thus, *Psets* are the observable properties. We employ only finitely supported ultrafilters, radically changing the resulting topology.

The aim of the work is to investigate a duality between the resulting topological spaces and permutation-boolean algebras, of the same kind of the fundamental duality holding between (finitary) boolean algebras and *Stone Spaces*.

The expected result is that the freshness quantifier of Pitts and Gabbay is required to complete the logic. The main point of interest is the following: it is well known that the full nominal logic, featuring quantification over names and syntactic freshness $x\#\phi$ is incomplete. Intuitively, the freshness quantifier $Nx.\phi$ allows one to check wether there *exists* a fresh name n' in a process p such that p models $\phi[n'/n]$. On the other hand, the freshness relation $x\#\phi$ allows one to check if x is actually fresh in p, and ϕ is satisfied by it. So, the freshness quantifier should be strictly weaker than the freshness relation. We aim at clarifying this point by finding the Stone-dual of permutation-boolean algebras.

2 Some technical remarks

Adding just a few more details, we seek for a pair of functors P and S

$$PBA \underset{P}{\overset{S}{\rightleftarrows}} TFS^{op}$$

giving rise to a duality between boolean-permutation algebras and those topological spaces TFS whose points form a finitely supported nominal set, i.e., such that there exist two natural isomorphisms of type $Id_{PBA} \to P \circ S$ and $S \circ P \to Id_{TFS}$. In Stone duality, given a boolean algebra A, typically the points of the space $S(A)$ are morphisms $A \to 2$: finitely supported, *maximal* and *consistent* collections of elements of A, that is, subobjects of A. In permutation algebras, this is no longer sufficient since the subobjects are all zero-supported. Instead, to recover a duality we have to add all the finitely-supported ultrafilters to the points of the topology.

On the other hand, elements of the algebra $P(A)$ for each space T are the finitely supported clopens (sets that are both open and closed) of the topology. As points in space are canonical models, the clopens represent properties.

On the side of topological spaces, hence of models, we examine the meaning of the formula $\bigvee_{\rho \in R} \rho(\phi)$, for $\phi \in A$ and R set of permutations sending *some* of the finite names of ϕ into *all* the possible names outside of the support of ϕ. Being infinitary, this formula does not belong to the basic logic: it generalizes the semantics of the freshness quantifier to an arbitrary number of names. This formula can be represented just as $Nx_1, \ldots, x_n.\phi$, where $x_1, \ldots, x_n$ are the names of ϕ that are permuted in all possible ways, viewed as *bound* and α-convertible names, thus adding a generalized freshness quantifier N to the signature of boolean-permutation algebras. We provide a model-theoretical proof of the existential/universal nature of the freshness quantifier that is known from Gabbay and Pitts. The remaining step is to prove the existence of topological spaces in TFS where the set of models of $Nx.\phi$ is a clopen, hence an observable property, but the set of models of $x\#\phi$ is not. The resulting logic, which is simpler than full nominal logic and the derived theories that include syntactic freshness, might prove useful in situations where syntactic names are too strong

as a property, since *observable* names are less than the syntactic ones. This phenomenon happens e.g. in the set of early input transitions of the π-calculus, where we find *redundant* names, that is, not observable in the final semantics.

References

1. Gabbay, M., Pitts, A.: A new approach to abstract syntax with variable binding. Formal Aspects of Computing **13** (2002) 341–363
2. Montanari, U., Pistore, M.: Structured coalgebras and minimal hd-automata for the π-calculus. Theoretical Computer Science **340** (2005) 539–576
3. Pistore, M.: History Dependent Automata. PhD thesis, Università di Pisa, Dipartimento di Informatica (1999)
4. Gadducci, F., Miculan, M., Montanari, U.: About permutation algebras, (pre)sheaves and named sets. Higher-Order and Symbolic Computation **19** (2006) 283–304
5. Caires, L., Cardelli, L.: A spatial logic for concurrency (part I). Information and Computation **186** (2003) 194–235

On spatio-temporal logics for the verification of structured interactive programs with registers and voices*

Cezara Dragoi and Gheorghe Stefanescu*****

Faculty of Mathematics and Computer Science, University of Bucharest
Str. Academiei 14, Bucharest, Romania 010014
{cdragoi,gheorghe}@funinf.cs.unibuc.ro

Abstract. Interactive programs with registers and voices (*rv-programs*) are an interactive computing model based on register machines and a space-time duality [1]. AGAPIA v0.1 [2] is a high-level structured programming language developed on top of rv-programs. In AGAPIA v0.1 one can write programs for open processes located at various sites and having their temporal windows of adequate reaction to the environment. The language naturally support process migration, structured interaction, and deployment of modules on heterogeneous machines.

In this paper we introduce a sound Hoare-like spatio-temporal logic for the verification of AGAPIA programs. As a case study, the verification of a termination detection protocol for a pool of distributed processes is presented.

1 Rv-systems and AGAPIA programming

A model (consisting of *rv-systems*), a core programming language (for developing *rv-programs*), several specification and analysis techniques appropriate for modeling, programming and reasoning about interactive computing systems have been introduced in [1] using register machines and a space-time duality.

In [2, 3], structured programming techniques for rv-systems have been introduced with a particular emphasis on developing a structural spatial programming discipline. This structured interaction between processes simplify the construction and the analysis of interactive programs. Compared with other interaction calculi, e.g., with π-calculus [4] or with actor models [5], our approach gives a name-free calculus.

* This research was partially supported by the Romanian Ministry of Education and Research (PNCDI-II Program 4, Project D1/1052/18.09.2007: *GlobalComp - Models, semantics, logics and technologies for global computing*). A draft of the paper may be found at www.cs.uiuc.edu/homes/stefanes/drafts/stLog08.pdf

** Current address: LIAFA, Universite Paris Diderot - Paris 7, France

*** Current address: Department of Computer Science, University of Illinois at Urbana-Champaign, USA

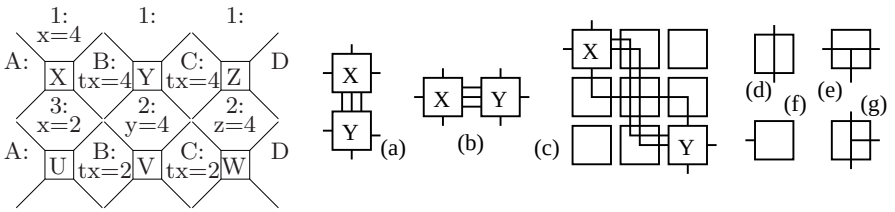


Fig. 1. A scenario and operations on scenarios.

Interfaces

$SST ::= nil \mid sn \mid sb \mid (SST \cup SST) \mid (SST, SST) \mid (SST)^*$

$ST ::= (SST) \mid (ST \cup ST) \mid (ST; ST) \mid (ST;)^*$

$STT ::= nil \mid tn \mid tb \mid (STT \cup STT) \mid (STT, STT) \mid (STT)^*$

$TT ::= (STT) \mid (TT \cup TT) \mid (TT; TT) \mid (TT;)^*$

Expressions

$V ::= x : ST \mid x : TT \mid V(k) \mid V.k \mid V.[k] \mid V@k \mid V@[k]$

$E ::= n \mid V \mid E + E \mid E * E \mid E - E \mid E/E$

$B ::= b \mid V \mid B\&\&B \mid B||B \mid !B \mid E < E$

Programs

$W ::= nil \mid new\ x : SST \mid new\ x : STT \mid x := E \mid if(B)\{W\}else\{W\} \mid W; W \mid while(B)\{W\}$

$M ::= module\{listen\ x : STT\}\{read\ x : SST\} \{W;\}\{speak\ x : STT\}\{write\ x : SST\}$

$P ::= nil \mid M \mid if(B)\{P\}else\{P\} \mid P\%P \mid P\#P \mid P\$P \mid while\_t(B)\{P\} \mid while\_s(B)\{P\} \mid while\_st(B)\{P\}$

Fig. 2. The syntax of AGAPIA v0.1 programs

A key feature which help in getting a structured interaction model is the throughly extension of the temporal data types used on interaction interfaces. These new temporal data types (including voices as a time-dual version of registers) are implemented on top of streams.

AGAPIA [2, 6] (see Fig. 2) is a kernel high-level massively parallel, interactive programming language. The language is natural and expressive, for instance one can easily model the activity of a ring of processes in an open environment where processes may freely join of leave the ring. The language has simple denotational and operational semantics based on scenarios. (Scenarios are a two-dimensional extension of the running paths used in imperative programs, see Fig. 1.) The language naturally support process migration, structured interaction, and deployment of modules on heterogeneous machines.

2 Towards a Hoare-like logic for structured rv-programs

In this paper we introduce a Floyd/Hoare-like verification logics for AGAPIA or, more generally, structured rv-programs. We present a set of sound rules for proving the correctness of structured rv-programs. As a case study, we present an implementation and a detailed formal verification of a popular distributed ter-

mination detection protocol. The method may be applied to many sophisticated distributed protocols.

A framework for rv-programs verification. For sequential programs one has to find assertions in a few key points of the program and to prove the invariance conditions. This technique demands to have at least one cut-point along each loop. The set of cut-points ensures that: (1) each syntactically possible path from input to output is decomposed by cut-points into a sequence $(p_1, p_2, \ldots, p_k)$ and (2) the set $SPath$ of all such paths p_i forms a *finite set*. The proof finally reduces to the verification of the invariance conditions for $SPath$.

For rv-programs, cut-points becomes contours, surrounding finite scenarios. Their set must be finite. The condition to "break all loops" becomes "each syntactically possible scenario can be decomposed into pieces corresponding to these contours".

To conclude, the verification procedure for rv-programs consists of the following three steps: (i) find an appropriate set of contours and assertions; (ii) fill in the contours with all possible scenarios; and (iii) prove these scenarios respect the border assertions. Notice that, except for the guess of assertions, the proof is finite and can be fully automatized.

Structured rv-programs have a more restricted way to construct scenarios, hence the procedure is more regular: one has to provide assertions for each statement and to lift them to the full program applying inference rules.

References

1. Stefanescu, G.: Interactive systems with registers and voices. Fundamenta Informaticae **73**(1-2) (2006) 285–305
2. Dragoi, C., Stefanescu, G.: Agapia v0.1: A programming language for interactive systems and its typing system. In Goldin, D., Arbab, F., eds.: Foundations of Interactive Computation. Volume 203 of Electronic Notes in Theoretical Computer Science., Elsevier (2008)
3. Dragoi, C., Stefanescu, G.: On compiling structured interactive programs with registers and voices. In Geffert, V., Karhumäki, J., Bertoni, A., Preneel, B., Návrat, P., Bieliková, M., eds.: SOFSEM. Volume 4910 of Lecture Notes in Computer Science., Springer (2008) 259–270
4. Milner, R.: Communicating and Mobile Systems: The Pi Calculus. Cambridge University Press (1999)
5. Agha, G.: Actors: A model of concurrent computation in distributed systems. MIT Press (1986)
6. Popa, A., Sofronia, A., Stefanescu, G.: High-level structured interactive programs with registers and voices. Journal of Universal Computer Science **13**(11) (2007) 1722–1754

The Van-Kampen Square in view of the Grothendieck construction*

Uwe Wolter[1] and Zinovy Diskin[2]

[1] Department of Informatics, University of Bergen
[2] Department of Computer Science, University of Toronto

Van-Kampen squares are the essential ingredient in the definition of Adhesive Categories as they have been introduced recently by Lack and Sobociński [1], and adhesive categories are considered as a promising formal basis for a general theory of transformations and of reactive systems. [2], for example, provides a thorough revision and generalization of the different variants of graph-transformations and high level replacement systems based on the concept of adhesive category.

We want to present and discuss here an observation we made when developing the semantics of the Generalized Skech (GS) framework, since we are convinced that this observation may be of wider interest.

The GS framework is a graph-based specification framework that borrows its main ideas from both categorical and first-order logic, and adapts them to software engineering needs. Following the "algebraic tradition" the semantics of the GS framework was defined first in an "indexed manner", i.e., models where defined as interpretations of specifications in a semantic universe as the categories Set, Graph, or Cat, for example [3, 4]. As one can expect, such an indexed semantics allows us, for example, to define the "amalgamated sums" of models by constructing the unique mediating morphism for a pushout diagram of specifications (compare [5]).

Essential parts of Software Engineering, however, are not based on indexed but on fibred semantics, i.e., on the concept of an instance of a specification. A given indexed semantics can be transformed into a corresponding fibred semantics when the underlying semantic universe allows for (a variant of) the so-called Grothendieck construction.

It is folklore that the Grothendieck construction turns composition, i.e., commutative triangles, into pullback diagrams. Analyzing the transition from indexed semantics in [4] to fibred semantics in [6] we observed in [7] that the existence of a unique mediating morphism for a pushout of specifications is turned by the Grothendieck construction into the task to construct a "unique pullback completion" for a half cube consisting of a pushout and two pullbacks. Or to formulate our observation as a slogan:

The Grothendieck construction transforms pushouts into "weak" van Kampen squares.

After presenting our observation for the simple framework with graphs as specifications, we intend to close our talk with a discussion of counter examples

* Research partially supported by the Norwegian NFR project SHIP.

for van-Kampen squares and a discussion of some open questions concerning the definition of van-Kampen squares.

References

1. Lack, S., Sobociński, P.: Adhesive Categories. In Walukiewicz, I., ed.: proceedings of FOSSACS 2004, Springer, LNCS 2987 (2004) 273–288
2. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformations. EATCS Monographs on Theoretical Computer Science. Springer, Berlin (2006)
3. Diskin, Z.: Databases as diagram algebras: Specifying queries and views via the graph-based logic of skethes. Technical Report 9602, Frame Inform Systems, Riga, Latvia (1996) http://citeseer.ist.psu.edu/116057.html.
4. Wolter, U., Diskin, Z.: The Next Hundred Diagrammatic Specification Techniques – An Introduction to Generalized Sketches. Technical Report Report No 358, Department of Informatics, University of Bergen (July 2007)
5. Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specification 1: Equations and Initial Semantics. Volume 6 of EATCS Monographs on Theoretical Computer Science. Springer, Berlin (1985)
6. Diskin, Z., Wolter, U.: Generalized Sketches: A Universal Logic for Diagrammatic Modeling in Software Engineering. ENTCS (2007) Accepted.
7. Wolter, U., Diskin, Z.: From Indexed to Fibred Semantics – The Generalized Sketch File. Technical Report Report No 361, Department of Informatics, University of Bergen (October 2007)

C-semiring Frameworks for MST and ST problems

Stefano Bistarelli[1,2], and Francesco Santini[2,3]

[1] Dipartimento di Scienze, Università "G. D'Annunzio" di Chieti-Pescara, Italy
bista@sci.unich.it,cmeo@unich.it
[2] Istituto di Informatica e Telematica (CNR), Pisa, Italy
[stefano.bistarelli,francesco.santini]@iit.cnr.it
[3] IMT - Istituto di Studi Avanzati, Lucca, Italy
f.santini@imtlucca.it

Introduction. Classical *Minimum Spanning Tree* (MST) problems [1] in a weighted directed graph arise in various contexts. One of the most immediate examples is related to the multicast communication scheme in networks with QoS requirements [2]. For example, we could need to optimize the bandwidth, the delay or a generic cost of the distribution towards several final receivers. In the same way, considering the *Steiner Tree* (ST) problem [3], given a set $S \subseteq V$ of vertices in a graph $G = (V, E)$, a solution interconnects them by a tree of minimum weight, where this weight is the sum of the weights of all the tree edges, represented by QoS metric values (e.g. cost and delay). If $S = V$, the ST problem reduces to the MST problem. ST is well-known NP-Complete problem [4].

A general algebraic framework for computing these problems has not been already studied. We propose to achieve this result by considering c-semirings structures.

C-semirings. A c-semiring S is a tuple $\langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$ where A is a set with two special elements ($\mathbf{0}, \mathbf{1} \in A$) and with two operations $+$ and $\times$ [5]: $+$ is defined over (possibly infinite) sets of elements of A and thus is commutative, associative, idempotent, it is closed and $\mathbf{0}$ is its unit element and $\mathbf{1}$ is its absorbing element; $\times$ is closed, associative, commutative, distributes over $+$, $\mathbf{1}$ is its unit element, and $\mathbf{0}$ is its absorbing element. The $+$ operation defines a partial order $\leq_S$ over A such that $a \leq_S b$ iff $a + b = b$; we say that $a \leq_S b$ if b represents a value *better* than a. Other properties related to the two operations are that $+$ and $\times$ are monotone on $\leq_S$, $\mathbf{0}$ is its minimum and $\mathbf{1}$ its maximum, $\langle A, \leq_S\rangle$ is a complete lattice and $+$ is its lub. Finally, if $\times$ is idempotent, then $+$ distributes over $\times$, $\langle A, \leq_S\rangle$ is a complete distributive lattice and $\times$ its glb [5].

Our goal. In our study we would like to define a general algebraic framework for MST and ST problems based on the structure of c-semirings. We want to give generic algorithms for solving these problems. The algorithm is generic in the sense that it has to work with any c-semiring covered by our general framework, where different c-semirings are used to model different QoS metrics.

Classical MST/ST problems can be generalized to other weight sets, and to other operations.

The weight of a tree is obtained by "multiplying" the edge weights along that tree by using the $\times$ semiring operator, and the min-weight tree from a node p to a set of destination nodes D is the "sum" of the weights of all the trees reaching from p, obtained by using the $+$ semiring operator. By varying the set A and the meaning of the $+$ and $\times$ operations, we can represent many different kinds of problems, having features like fuzziness, probability, and optimization [5]. Notice also that the cartesian product of two c-semirings is a c-semiring [5], and this can be fruitfully used to describe multi-criteria optimization problems.

Moreover, in order to detail the framework, we want to prove the correctness and termination of this kind of algorithms, including a full analysis of its running time complexity with respect to the times to compute the $+$ and $\times$ c-semiring operations.

A similar framework was already provided in [6], but it covers shortest-path problems only. Therefore, we wish to extend it by considering also tree-related structures over graphs (MST and ST problems). Notice that, while in [6] the author uses also semirings with a non-idempotent $+$, we would like to focus mainly on c-semirings instead. The reason is that one more ambition could be to merge these kind of frameworks with routing constraints concerning the considered QoS metrics (e.g. $delay \leq 40$ or $minimize(cost)$), since *Soft Constraint Satisfaction Problems* based on c-semirings have been already successfully applied to this field [7, 8].

A first sketch of a possible algorithm for a MST problem over a graph $G(V, E)$ is given in Alg. 1. It is obtained by modifying the classical Kruskal algorithm [1] in order to use c-semirings values and operators which are taken as input, i.e. $\langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$. To find the (partial) solution tree T, the complexity of the algorithm is $O(|E|)$ as in the original procedure. b is the best edge in the current iteration of the *repeat* command (row 2) and it is found (in row 3) by applying the $\bigoplus$ operator over all the remaining edges in the set P (i.e. the set of possible edges), instantiated to E at the beginning (row 1); $\bigoplus$ finds the best edge according to the ordering defined by the $+$ operator of the semiring. Then the (partial) solution tree is updated with the $\bigotimes$ operator, which adds the new edge and updates the cost of the tree according to the $\times$ operator of the semiring (row 5). At last, b is removed from P (row 7).

We can show also that the other best-known algorithm for solving the MST problem can be generalized with semiring structures (see Alg. 2). Step by step, the modified Prim's algorithm [1] adds an edge to the (partial solution) tree T, instead of joining a forest of trees in a single connected tree, as in Kruskal. However, even Prim's procedure proceeds in a greedy way by choosing the best-cost edge (i.e. (v_i, u_j) in row 3) in order to add it to the solution (row 4).

Since some heuristics used to find solutions for the ST problems are based on discovering a MST solution for the same graph [3], these results could be extended to ST problems as well. Notice also that Alg. 1 and Alg. 2 works only if the set of costs is totally ordered, while it needs to be modified otherwise for

Algorithm 1 `Kruskal with semiring structures`

INPUT: $G(V,E)$, $\langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$

1: $T = \emptyset, P = E$
2: **repeat**
3: $\quad b = \bigoplus_{\forall e \in P}(e)$
4: $\quad$ **if** (endpoints of b are disconnected in T) **then**
5: $\quad\quad T = T \bigotimes b$
6: $\quad$ **end if**
7: $\quad P = P \setminus \{b\}$
8: **until** $P = \emptyset$

OUTPUT: $T \equiv MST$ over G

a multicriteria optimization, since the costs of the edges are partially ordered. In this case, the semiring operators have to deal with multiset of solutions that are Pareto-optimal.

Algorithm 2 `Prim with semiring structures`

INPUT: $G(V,E)$, $\langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$

1: $T = \emptyset$, $R = \{v_k\}$, v_k is arbitrary
2: **repeat**
3: $\quad (v_i, u_j) = \bigoplus_{\forall e \in P}(e)$ *s.t.* $P = \{(v_k, u_z) \in E \,|\, (v_k \in R) \wedge (u_z \notin R)\}$
4: $\quad T = T \bigotimes b$
5: $\quad R = R \cup u_j$
6: **until** $R = V$

OUTPUT: $T \equiv MST$ over G

References

1. Cormen, T.T., Leiserson, C.E., Rivest, R.L.: Introduction to algorithms. MIT Press, Cambridge, MA, USA (1990)
2. Wang, B., Hou, J.: Multicast routing and its QoS extension: problems, algorithms, and protocols. IEEE Network **14** (January 2000)
3. Kou, L.T., Markowsky, G., Berman, L.: A fast algorithm for steiner trees. Acta Inf. **15** (1981) 141–145
4. Winter, P.: Steiner problem in networks: a survey. Netw. **17**(2) (1987) 129–167
5. Bistarelli, S.: Semirings for Soft Constraint Solving and Programming. Volume 2962 of Lecture Notes in Computer Science. Springer, London, UK (2004)
6. Mohri, M.: Semiring frameworks and algorithms for shortest-distance problems. J. Autom. Lang. Comb. **7**(3) (2002) 321–350
7. Bistarelli, S., Montanari, U., Rossi, F.: Soft constraint logic programming and generalized shortest path problems. Journal of Heuristics **8**(1) (2002) 25–41
8. Bistarelli, S., Montanari, U., Rossi, F., Santini, F.: Modelling multicast QoS routing by using best-tree search in and-or graphs and soft constraint logic programming. Electr. Notes Theor. Comput. Sci. **190**(3) (2007) 111–127

A term-graph syntax for algebras over multisets

Fabio Gadducci

Dipartimento di Informatica, Università di Pisa
Polo "Guglielmo Marconi", via dei Colli 90, 19100 La Spezia, Italia

Abstract. Earlier papers [1, 2] argued that term graphs play for the specification of relation-based algebras the same role that standard terms play for total algebras. The present contribution enforces the claim, by proving that term graphs are a sound and complete representation for algebras whose operators are interpreted over multisets.

1 Introduction

Cartesian categories offer the right tool for interpreting equational logic: objects are tuples of sorts, and arrows are tuples of terms, typed accordingly. This is confirmed by the presentation of the category of (total) algebras for a signature Σ as the category of product-preserving functors from the cartesian category $\mathbf{Th}(\Sigma)$, the *algebraic theory* of Σ, to the category **Set** of sets and functions.

Two arrows in $\mathbf{Th}(\Sigma)$ coincide iff they denote the same function for every functor. Thus, equational signatures (Σ, E) and their categories of algebras can be easily recast in the framework by quotienting the arrows corresponding to pairs of terms $(s, t) \in E$ (see [3] for an introduction to the topic). Moreover, standard equational logic rules correspond to arrow composition in $\mathbf{Th}(\Sigma)$.

Such a characterisation proved elusive for more complex algebraic formalisms, such as partial algebras and especially multialgebras [4], where operators are interpreted as partial functions and as additive relations, respectively. The main result of [1] is the introduction of suitable categories for a signature Σ (based on the gs-monoidal theory $\mathbf{GS\text{-}Th}(\Sigma)$ of Σ, generalizing $\mathbf{Th}(\Sigma)$) for obtaining a functorial representation of the categories of partial and multialgebras for Σ.

The solution proved satisfactory for partial algebras, since the arrows of a quotient category of $\mathbf{GS\text{-}Th}(\Sigma)$ (namely, the g-monoidal theory $\mathbf{G\text{-}Th}(\Sigma)$) are in bijective correspondence with conditioned terms, i.e., with pairs $s \mid D$, where s is the principal term and D is a set of terms used for restricting the domain of definition of s. Most importantly, two arrows in $\mathbf{G\text{-}Th}(\Sigma)$ coincide iff they always denote the same partial function for every possible functor, thus allowing the development of a sound and complete deduction system for equational signatures based on so-called conditioned Kleene equations [5].

Things went less smoothly for multialgebras. Indeed, the arrows of the gs-monoidal theory $\mathbf{GS\text{-}Th}(\Sigma)$ are in bijective correspondence with (acyclic) term graphs, i.e, trees with possibly shared nodes. However, as shown in [2], only term graphs up-to *garbage equivalence* are identified by every functor. Such an equivalence is defined set-theoretically, and an axiomatic presentation is missing.

The present paper shows that the gs-monoidal theory **GS-Th(Σ)** allows for a functorial presentation of what we called *multiset algebras*, that is, algebras whose operators are interpreted as (additive) multiset relations. Furthermore, we prove that two term graphs denote the same multiset relation iff they are isomorphic, thus laying the base for a simple deduction system for such algebras.

2 Basic notions of graphs and relations

This section presents some definitions concerning term graphs with interfaces, as well as (additive) multiset relations. We refer to [6] and [1] for an introduction.

Definition 1 (Graphs). *A graph is a 4-tuple $\langle V, E, s, t\rangle$ where V is the set of nodes, E is the set of edges and $s, t : E \to V^*$ are the source and target functions.*

A typed graph *G over T is a graph $|G|$ with a graph morphism $\tau_G : |G| \to T$.*

Let J, K be typed graphs. A graph with input interface J and output interface K is a triple $\mathbb{G} = \langle j, G, k\rangle$, where G is a typed graph, $j : J \to G$ and $k : K \to G$ are injective and they are called input and output morphisms, respectively.

Graphs and typed graph morphisms are defined intuitively. In the following, a signature Σ is thus a graph such that the source function takes value in V; while a term graph over Σ is an acyclic graph typed over Σ such that each node is in the image of the source function of at most one node.

Definition 2 (Relations). *Let X, Y be sets. A multiset relation is a function from X to $[Y \to \mathbb{N}]_f$, i.e., associating to each element $x \in X$ a function h (with finite support) from Y to the natural numbers.*

"Finite support" means that $h(y) \neq 0$ for a finite set of elements y. A partial function requires that $h(y) = 1$ for at most one element y, while $h(z) = 0$ for all the others. An additive relation is described by replacing $\mathbb{N}$ with the set $\{0, 1\}$.

3 Categories with a gs-monoidal structure

This section recalls the definition of gs-monoidal category, and states the correspondence between arrows of a gs-monoidal category and term graphs [6].

A category **C** is gs-monoidal if it is equipped with an operator $X \otimes Y$, making it a symmetric strictly monoidal category [7] with symmetry ρ and unit λ, and two morphisms $\nabla_X : X \to X \otimes X$ and $!_X : X \to \lambda$ such that each object is a comonoid object [8] (equivalently, that the axioms of Fig. 1 hold). Should the morphisms be natural, the resulting category would be cartesian.

Mimicking the correspondence between terms and trees, morphisms of a gs-monoidal category correspond to term graphs [6]. The lack of naturality allows the distinction between sharing a term and the occurrence of two copies of it.

Proposition 1. *Let Σ be a signature. The arrows of the gs-monoidal theory* **GS-Th(Σ)** *(the free gs-monoidal category over Σ) are in bijective correspondence with the term graphs with discrete and ordered interfaces, typed over Σ.*

An interface must then contain no edge, and its nodes be totally ordered.

$$!_{X\otimes Y} = !_X \otimes !_Y \qquad !_\lambda = \nabla_\lambda = id_\lambda \qquad \nabla_{X\otimes Y} = (id_X \otimes \rho_{X,Y} \otimes id_Y) \circ (\nabla_X \otimes \nabla_Y)$$

$$(id_X \otimes !_Y) \circ \nabla_X = id_X \qquad \rho_{x,X} \circ \nabla_X = \nabla_X \qquad (\nabla_X \otimes id_X) \circ \nabla_X = (id_X \otimes \nabla_X) \circ \nabla_X$$

Fig. 1. Axioms for gs-monoidality.

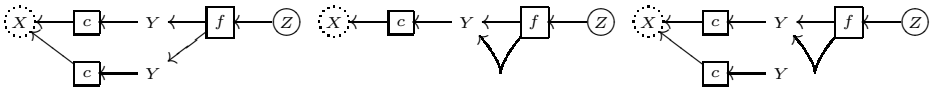


Fig. 2. Graphs F_1, F_2, and F_3.

Consider the term graphs in Fig. 2 (from [1, Fig. 2]): nodes in the input (output) interface are circled with a dotted (solid) circle. The signature contains a unary c and a binary f, and nodes and edges are labelled by their type. F_1, F_2 and F_3 correspond to arrows $f \circ (c \otimes c) \circ \nabla_X$, $\alpha = f \circ \nabla_Y \circ c$ and $(\alpha \otimes (!_Y \circ c)) \circ \nabla_X$, respectively, belonging to hom-set **GS-Th(Σ)**$[X, Z]$. The first two arrows differ because ∇ is not natural, and the latter two because ! is not natural.

4 Interpreting graphs over multisets

The first step is defining a category **M-Rel** of multiset relations. From now on, a multiset is denoted in polynomial form, so $n_1 \cdot y_1 \oplus \ldots \oplus n_k \cdot y_k$ is the function associating number n_i to y_i, and 0 to the other elements (assuming y_i's pairwise different). So, for relations $h : X \to Y$, $k : X \to W$ and $h_1 : Y \to Z$, we define

- $\forall x \in X.\, h_1 \circ h(x) = \bigoplus_{n \cdot y \in h(x)} n \cdot h_1(y)$;
- $\forall xw \in X^2.\, h \otimes k(xw) = \bigoplus_{n \cdot y \in h(x), m \cdot z \in k(w)} nm \cdot yz$.

Products and sums on the scalar components of a polynomial are defined in the obvious way. Indeed, with such a choice the resulting monoidal category **M-Rel** is the Kleisli category induced by the monad associating to a set X the set of finite polynomials over X (i.e., equivalently, the finitely supported function $X \to \mathbb{N}$), with the monoidal tensor $\otimes$ lifting the cartesian product of **Set**.

The category is indeed gs-monoidal, for $!_X$ associating to each $x \in X$ the empty multiset; and ∇_X associating to each $x \in X$ the one-element multiset xx. At this point we can map the arrows of the gs-monoidal theory over **M-Rel**.

Theorem 1. *Let Σ be a signature, and $s, t : X \to Y$ arrows in* **GS-Th(Σ)**. *Then, s and t represent isomorphic term graphs iff they are mapped to the same multiset relation for each gs-monoidal functor from* **GS-Th(Σ)** *to* **M-Rel**.

A gs-monoidal functor maps the tensor and the ∇ and ! arrows in the expected way. The category [**GS-Th(Σ)**, **M-Rel**] of gs-monoidal functors offers a faithful description for multiset algebras and point-to-point homomorphisms.

The result has deeper implications. Indeed, even if the category [**GS-Th(Σ)**, **Rel**] faithfully represents multialgebras [1], functorial equivalence was not captured axiomatically, thus forbidding the development of an equational deduction system (only partly solved by the inequational one presented in [2]).

Instead, the arrows of **G-Th(Σ)** (the category obtained by quotienting **GS-Th(Σ)** with respect to axioms $\nabla_Y \circ f = (f \otimes f) \circ \nabla_X$ for all $f \in \Sigma$) represent conditioned terms, and [**G-Th(Σ)**, **PFun**] and [**GS-Th(Σ)**, **PFun**] turn out to be equivalent: this allows the development of a sound and complete deduction system for partial algebras based on conditioned Kleene equations.

The correspondence established by the above theorem suggests that term graphs are an adequate syntax on which to build an equational deduction system, and possibly an inequational one, for multiset algebras. The expressiveness of such calculi, of course, has yet to be fully explored.

Let us now look at the examples. As arrows of the g-monoidal category **G-Th(Σ)**, the three term graphs of Fig. 2 represent the same (possibly) partial function, corresponding to the conditioned term $f(c(x), c(x)) \mid \emptyset$. While F_2 and F_3 coincide also as additive relations, since garbage equivalent (according to [1]).

In order to ease the presentation, consider the multiset algebra $X = \{x\}$, $Y = \{y_1, y_2\}$ and $Z = \{z_{ij} \mid i = 1, 2\}$, and interpret the operators as $c(x) = \bigoplus_{i=1,2} n_i \cdot y_i$ and $f(y_i y_j) = z_{ij}$ for $i, j = 1, 2$. We now consider the derived operation $c \otimes c$, which evaluates xx to $\bigoplus_{i,j=1,2} n_i n_j \cdot y_i y_j$. We leave to the reader, by exploiting their arrow representation, to check that F_1 maps x to $\bigoplus_{i,j=1,2} n_i n_j \cdot z_{ij}$, F_2 maps x to $\bigoplus_{i=1,2} n_i \cdot z_{ii}$ and F_3 maps x to $\bigoplus_{i,j=1,2} n_i n_j \cdot z_{ii}$.

So, the three term graphs differ as multisets. We confirm that they concide as partial functions: if at most one between n_1, n_2 is 1, and the other is 0, then the three expressions coincide and they evaluate to either 0, z_{11} or z_{22}.

The situation is more complex when interpreting the expressions as additive relation. It means to replace the natural numbers s with the boolean algebra $\{0, 1\}$, thus interpreting $\oplus$ as $\cup$. Thus, F_1 is then evaluated to the set Z, while F_2 and F_3 are evaluated to the set $\{z_{11}, z_{22}\}$.

References

1. Corradini, A., Gadducci, F.: A functorial semantics for multi-algebras and partial algebras, with applications to syntax. Theor. Comp. Sci. **286** (2002) 293–322
2. Corradini, A., Gadducci, F., Kahl, W., König, B.: Inequational deduction as term graph rewriting. In Mackie, I., Plump, D., eds.: Term Graph Rewriting. Volume 72.1 of Electr. Notes in Theor. Comp. Sci., Elsevier (2002) 31–44
3. Pitts, A.M.: Categorical Logic. In Abramsky, S., Gabbay, D.M., Maibaum, T.S.E., eds.: Handbook of Logic in Computer Science. Volume 6. Oxford University Press (2000)
4. Walicki, M., Meldal, S.: Algebraic approaches to nondeterminism: An overview. ACM Computing Surveys **29** (1997) 30–81
5. Burmeister, P.: Partial algebras - An introductory survey. In Rosenberg, I.G., Sabidussi, G., eds.: Algebras and Orders. NATO ASI Series C. Kluwer Academic (1993) 1–70
6. Corradini, A., Gadducci, F.: An algebraic presentation of term graphs, via gs-monoidal categories. Applied Categorical Structures **7** (1999) 299–331
7. Mac Lane, S.: Categories for the Working Mathematician. Springer (1971)
8. Fox, T.: Coalgebras and cartesian categories. Communications in Algebra **4** (1976) 665–667

Author Index