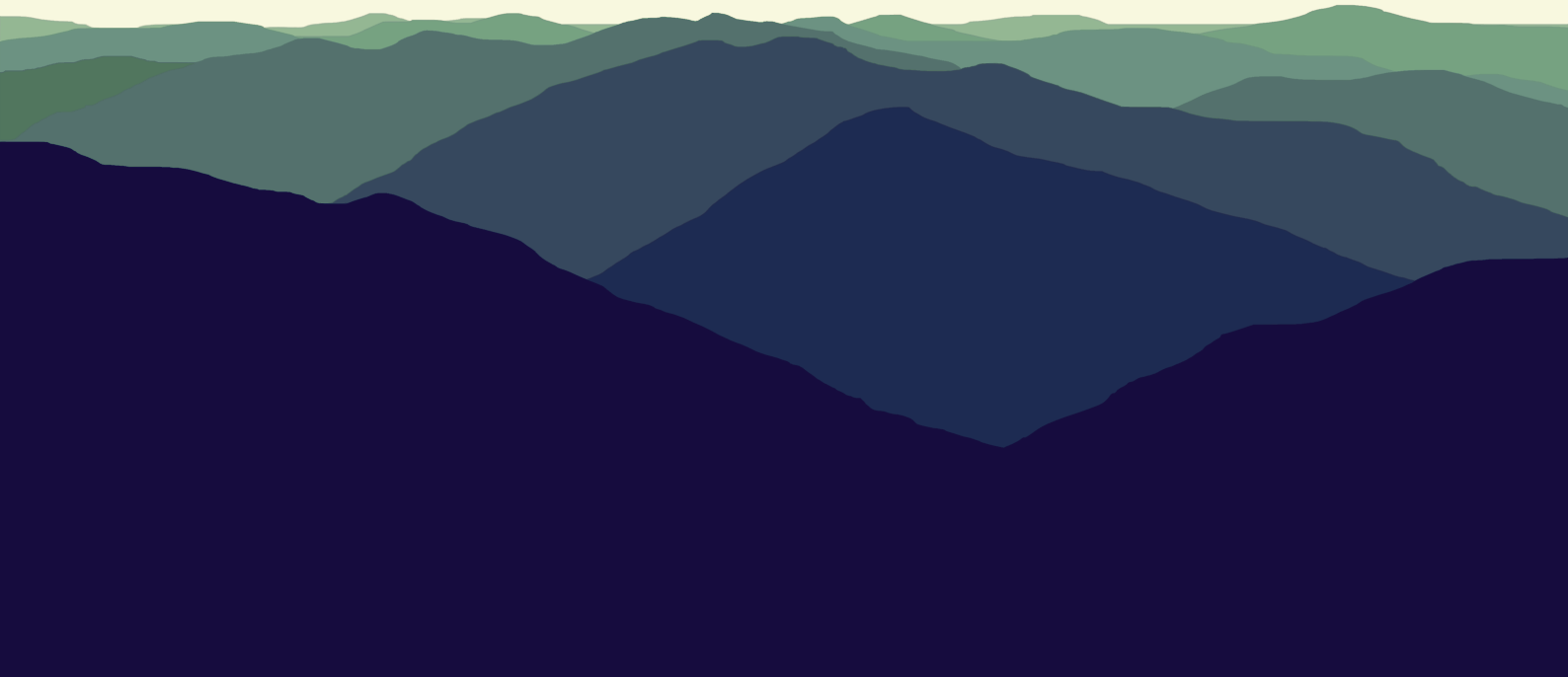
WADT 2014

22nd International Workshop on
Algebraic Development Techniques

SEPTEMBER 4-7, 2014
SINAIA, ROMANIA

In memoriam Joseph A. Goguen



WADT 2014
Preliminary Proceedings

22nd International Workshop on Algebraic Development Techniques

Răzvan Diaconescu Mihai Codescu Ionuţ Ţuţu

Technical report

Simion Stoilow Institute of Mathematics
of the Romanian Academy

September 2014

Preface

The 22nd International Workshop on Algebraic Development Techniques (WADT 2014) took place in Sinaia, Romania, from 4th to 7th September 2014. The event was dedicated to the memory of Joseph A. Goguen – one of the founding members of the ADT community – who visited Sinaia many times with great pleasure to meet with his former student and close friend Răzvan Diaconescu, the chair of WADT 2014. Sinaia is a beautiful mountain resort that has become a traditional Romanian venue for mathematical and theoretical computer science events, from the 2nd edition of the International Mathematical Olympiad (held in 1960) to the more recent Summer School on Language Frameworks and the Romanian-Japanese Algebraic Specification Workshops, which discussed some of the latest developments related to algebraic specification and in particular to the CafeOBJ language.

The algebraic approach to system specification encompasses many aspects of the formal design of software systems. Originally born as 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, concurrent, distributed and mobile systems). Thus, typical topics of interest are:

- foundations of algebraic specification,
- other approaches to formal specification, including 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, quality assurance, validation, and verification.

As 22 occurrences of the ADT Workshop can be considered as something to be noticed, a short look back may be allowed. The first workshop took place in 1982 in Sorpesee followed by Passau 1983, Bremen 1984, Braunschweig 1986, Gullane 1987, Berlin 1988, Wusterhausen 1990, Dourdan 1991, Caldes de Malavella 1992, S. Margherita 1994, Oslo 1995, Tarquinia 1997, Lisbon 1998, Chateau de Bonas 1999, Genova 2001, Frauenchiemsee 2002, Barcelona 2004, La Roche en Ardenne 2006, Pisa 2008, Etelsen 2010, and Salamanca 2012. With only a few exceptions at the beginning, it became also a tradition to publish selected papers after each workshop as a recent-trends series in Lecture Notes in Computer Science 332, 534, 655, 785,

906, 1130, 1376, 1589, 1827, 2267, 2755, 3423, 4409, 5486, 7137, and 7841, while the first volume of this kind appeared as Informatik-Fachberichte 116. This speaks for the stability of the ADT community and the continuity of the topics of interest. One should realize, however, that some significant transformation took place from 1982 to today. While ADT stood for *Abstract Data Types* at the beginning, it is now (since 1997) the acronym for *Algebraic Development Techniques*, and the list of topics has broadened in an amazing way.

This volume contains the 32 abstracts of ongoing research results presented during the workshop, including the two invited talks by K. Rustan M. Leino (Microsoft Research, USA) and Christoph Benzmüller (Freie Universität Berlin, Germany). As for previous ADT workshops, after the meeting authors will be invited to submit full papers for the refereed proceedings, which will be published as a volume of Springers Lecture Notes in Computer Science.

The WADT Steering Committee consists of:

- Michel Bidoit (France)
- Andrea Corradini (Italy)
- José Fiadeiro (UK)
- Rolf Hennicker (Germany)
- Hans-Jörg Kreowski (Germany)
- Till Mossakowski (Germany) – chair
- Fernando Orejas (Spain)
- Francesco Parisi-Presicce (Italy)
- Grigore Roşu (United States)
- Andrzej Tarlecki (Poland)

The Local Organizing Committee of WADT 2014 consists of former lecturers and students at the Postgraduate Academic Studies School "Şcoala Normală Superioară Bucureşti" (SNSB):

- Răzvan Diaconescu (Romania) – chair
- Mihai Codescu (Germany)
- Ionuţ Ţuţu (UK)

The workshop took place under the auspices of IFIP WG1.3 and benefited from the support offered by IFIP TC1 and the Simion Stoilow Institute of Mathematics of the Romanian Academy (IMAR).

September 2014
Sinaia

Răzvan Diaconescu
Mihai Codescu
Ionuţ Ţuţu

Contents

Invited talks

Regular presentations

On Logic Embeddings and Gödel's God

Christoph Benzmüller*

Freie Universität Berlin, Germany
c.benzmueller@fu-berlin.de

Logic embeddings provide an elegant means to formalize sophisticated non-classical logics in classical higher-order logic (HOL, Church's simple type theory [11]). In previous work (cf. [4] and the references therein) the embeddings approach has been successfully applied to automate object-level and meta-level reasoning for a range of logics and logic combinations with off-the-shelf HOL theorem provers. This also includes quantified modal logics (QML) [7] and quantified conditional logics (QCL) [3]. For many of the embedded logics few or none automated theorem provers did exist before. HOL is exploited in this approach to encode the semantics of the logics to be embedded, for example, Kripke semantics for QMLs [12] or selection function semantics for QCLs [23]. In some way, the indirect (relative to HOL) embeddings approach is orthogonal to labelled deductive systems [15], which employ (world-)labeling techniques for the direct modeling and implementation of non-classical proof systems.

In recent work [6, 5] we have applied the embeddings approach to verify and automate a philosophical argument that has fascinated philosophers and theologists for about 1000 years: the ontological argument for the existence of God [22]. We have thereby concentrated on Gödel's [16], respectively Scott's [21], modern version of this argument, which employs a second-order modal logic, for which, until now, no theorem provers were available. In our computer-assisted study of the argument, the HOL provers LEO-II [8] and Satallax [10] have made some interesting observations, some of which were unknown so far.

Ongoing and future work concentrates on the systematic study of Gödel's and Scott's proofs. We have also begun to study more recent variants of the argument [2, 1, 9, 14, 13, 17, 18], which claim to remedy some fundamental problem of Gödel's and Scott's proofs, known as the modal collapse. The long-term goal is to work out a landscape of the detailled logic parameters (e.g., constant vs. varying domains, rigid vs. non-rigid terms, logics KB vs. S4 vs. S5, etc.) under which the proposed variants of the modern ontological argument hold or fail.

There is little related work [19, 20], and this focuses solely on the comparably simpler, original ontological argument by Anselm of Canterbury.

Our work attests the maturity of contemporary interactive and automated deduction tools for HOL and demonstrates the elegance and practical relevance of the embeddings-based approach. Most importantly, our work opens new perspectives towards a computational metaphysics.

* This work has been supported by the German Research Foundation DFG under grants BE2501/9-1 & BE2501/11-1.

Acknowledgement: The study of Gödel's ontological proof of God's existence is joint work with Bruno Woltzenlogel Paleo.

References

1. A.C. Anderson and M. Gettings. Gödel ontological proof revisited. In *Gödel'96: Logical Foundations of Mathematics, Computer Science, and Physics: Lecture Notes in Logic 6*, pages 167–172. Springer, 1996.
2. C.A. Anderson. Some emendations of Gödel's ontological proof. *Faith and Philosophy*, 7(3), 1990.
3. C. Benzmüller. Automating quantified conditional logics in HOL. In F. Rossi, editor, Proc. of *IJCAI 2013*, pages 746–753, Beijing, China, 2013.
4. C. Benzmüller. A top-down approach to combining logics. In *Proc. of ICAART 2013*, pages 346–351, Barcelona, Spain, 2013. SciTePress Digital Library.
5. C. Benzmüller and B. Woltzenlogel Paleo. Gödel's God in Isabelle/HOL. *Archive of Formal Proofs*, 2013.
6. C. Benzmüller and B. Woltzenlogel Paleo. Automating Gödel's ontological proof of god's existence with higher-order automated theorem provers. In *ECAI 2014*, volume 263 of *Frontiers in Artificial Intelligence and Applications*, pages 163 – 168. IOS Press, 2014.
7. C. Benzmüller and L. Paulson. Quantified multimodal logics in simple type theory. *Logica Universalis (Special Issue on Multimodal Logics)*, 7(1):7–20, 2013.
8. C. Benzmüller, F. Theiss, L. Paulson, and A. Fietzke. LEO-II - a cooperative automatic theorem prover for higher-order logic (system description). In *Proc. of IJCAR 2008*, volume 5195 of *LNCS*, pages 162–170. Springer, 2008.
9. F. Bjørdal. Understanding gödel's ontological argument. In T. Childers, editor, *The Logica Yearbook 1998*. Filosofia, 1999.
10. C.E. Brown. Satallax: An automated higher-order prover. In *Proc. of IJCAR 2012*, number 7364 in LNAI, pages 111 – 117. Springer, 2012.
11. A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
12. M. Fitting and R.L. Mendelsohn. *First-Order Modal Logic*, volume 277 of *Synthese Library*. Kluwer, 1998.
13. M. Fitting. *Types, Tableaus, and Gödel's God*. Kluwer, 2002.
14. A. Fuhrmann. Existenz und Notwendigkeit — Kurt Gödels axiomatische Theologie. In W. Spohn et al., editor, *Logik in der Philosophie*. Heidelberg (Synchron), 2005.
15. D.M. Gabbay. *Labelled Deductive Systems*. Clarendon Press, 1996.
16. K. Gödel. *Appx.A: Notes in Kurt Gödel's Hand*, pages 144–145. In [22], 2004.
17. P. Hajek. A new small emendation of gödel's ontological proof. *Studia Logica: An International Journal for Symbolic Logic*, 71(2):pp. 149–164, 2002.
18. P. Hajek. Ontological proofs of existence and non-existence. *Studia Logica: An International Journal for Symbolic Logic*, 90(2):pp. 257–262, 2008.
19. P.E. Oppenheimera and E.N. Zalta. A computationally-discovered simplification of the ontological argument. *Australasian J. of Philosophy*, 89(2):333–349, 2011.
20. J. Rushby. The ontological argument in PVS. In *Proc. of CAV Workshop "Fun With Formal Methods"*, St. Petersburg, Russia,, 2013.
21. D. Scott. *Appx.B: Notes in Dana Scott's Hand*, pages 145–146. In [22], 2004.
22. J.H. Sobel. *Logic and Theism: Arguments for and Against Beliefs in God*. Cambridge U. Press, 2004.
23. R.C. Stalnaker. A theory of conditionals. In *Studies in Logical Theory*, pages 98–112. Blackwell, 1968.

An Interface to Symbolic Methods

K. Rustan M. Leino

Microsoft Research, Redmond, WA, USA
leino@microsoft.com

Symbolic methods that give computer-aided organization and detailed checking of proofs have become usable for many applications in mathematical domains. The Coq [2] and Isabelle/HOL [11] interactive proof assistants are prime examples of tools that rouinely support the writing of such proofs. While such systems provide unprecedented rigor in proofs, they are not always easy to use.

The last couple of decades have seen dramatic improvements in automatic satisfiability-modulo-theories (SMT) solvers [1]. These have been used in a large number of tools for software engineering and program analysis, where the logical conjectures to be resolved, albeit large, tend to be mathematically shallow. It seems, however, that such automatic techniques can also be applied to problems with more advanced mathematical contents and more difficult theorems.

In this talk, I will present a glimpse of what a more automatic interface to dealing with symbolic techniques can look like. I will do so by showing a number of examples using the Dafny language and verifier [4]. Built on top of an SMT solver and initially designed for verifying first-order imperative programs, Dafny now boasts support for induction [5], co-induction [7], and higher-order functions [9] and proofs can be structured into human-readable calculations [8]. Importantly, the tool's integrated development environment constantly runs the verifier in the background and helps the user in the development of proofs by keeping the available CPU cycles focused on what the user is working on [10].

For other examples, see, e.g., [6, 0,3].

References

0. Nada Amin. How to write your next POPL paper in Dafny. Lecture at Microsoft Research, July 2013. `http://research.microsoft.com/apps/video/default.aspx?id=198423`.
1. Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB standard: Version 2.0. In Aarti Gupta and Daniel Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories*, 2010.
2. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development — Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004.
3. K. Rustan M. Leino. Verification Corner. Channel on `youtube.com`. With various guests.
4. K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *LPAR-16*, volume 6355 of *LNCS*, pages 348–370. Springer, April 2010.
5. K. Rustan M. Leino. Automating induction with an SMT solver. In Viktor Kuncak and Andrey Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation — 13th International Conference, VMCAI 2012*, volume 7148 of *LNCS*, pages 315–331. Springer, January 2012.

6. K. Rustan M. Leino. Automating theorem proving with SMT. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving — 4th International Conference, ITP 2013*, volume 7998 of *LNCS*, pages 2–16. Springer, July 2013.
7. K. Rustan M. Leino and Michał Moskal. Co-induction simply - automatic co-inductive proofs in a program verifier. In Cliff B. Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014: Formal Methods — 19th International Symposium*, volume 8442 of *LNCS*, pages 382–398. Springer, May 2014.
8. K. Rustan M. Leino and Nadia Polikarpova. Verified calculations. In *Fifth Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE 2013)*, May 2013.
9. K. Rustan M. Leino and Dan Rosén. Higher-order functions in Dafny. Dan Rosén's research internship, Microsoft Research, August 2014.
10. K. Rustan M. Leino and Valentin Wüstholz. The Dafny integrated development environment. In Catherine Dubois, Dimitra Giannakopoulou, and Dominique Méry, editors, *Proceedings 1st Workshop on Formal Integrated Development Environment, F-IDE 2014*, volume 149 of *EPTCS*, pages 3–15, April 2014.
11. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

Integrating Athena with Algebraic specifications

Konstantine Arkoudas, Katerina Ksystra,
Nikos Triantafyllou, and Petros Stefaneas

Algebraic specification languages such as CafeOBJ [1], Maude [2] and CASL [3] have well-known advantages for modeling digital systems. The specifications are relatively simple, readable and writable, and can be executed and automatically analyzed in various other ways to provide valuable information to the modelers. In some cases, it is useful to attempt to prove that the model – usually couched as a transition system – has certain properties. For that purpose, some algebraic specification languages have been coupled with interactive theorem-proving systems; CASL, for instance, has been interfaced with HOL/Isabelle [4].

We propose a framework of integration of CafeOBJ [5,6] with Athena [7], a system based on general polymorphic multi-sorted first-order logic. Athena integrates computation and deduction, allows for readable and highly structured proofs, guarantees the soundness of results that have been proved, and also has built-in mechanisms for general model-checking and theorem-proving, as well as connections to state-of-the-art external systems for both. Our aim in this effort is to combine the strengths of CafeOBJ [8,9] (most notably succinct, composable, executable specifications based on conditional equational logic) with those of Athena, namely, structured and readable proofs, and greater automation both for proof and for counterexample discovery.

We illustrate our approach with a simple mutex algorithm. We demonstrate five aspects of formal methods: specification, simulation, automated counterexample generation, automated theorem proving, and automated checking of structured proofs in natural deduction style.

In our mutex example there is a set of processes, each of which is executing code. At any point in time (i.e., at any system state), a process is either in some *critical section* of the code or in some remainder (non-critical) section. When a process p enters its critical section, the resulting state becomes *locked*. When p exits the critical section, the resulting state is unlocked. For p to enter its critical section in some state s, p must be *enabled* in s. A process p is enabled in s iff p is in its remainder section in s and s is not locked. This is, therefore, the effective condition of the *enter* state transition for a given process. The effective condition of the *exit* transition is for the process to be in its critical section. We have two observer functions, one that takes a state s and a process id p and tells us what section of the code p is executing in s (critical or remainder), and a function that takes a state s and tells us whether s is locked. We present the specification of this algorithm in Athena and we prove that no more than one process can be in its critical section at any one time. A completely automatic proof by structural induction is obtained first, and then a more detailed – and more explanatory – proof in natural-deduction style is given and automatically checked for soundness. We also demonstrate a systematic method for simulating

the execution of the specified system that has been formalized in Athena, and for discovering counterexamples to invariant conjectures.

When rewriting fails during a proof obligation in a given "proof score" (i.e., when the two sides of a desired identity do not reduce to the same normal form by using all available equations as left-to-right rewrite rules), systems such as CafeOBJ will inform the user of how far the two sides could be rewritten, and this feedback often suggests lemmas that are necessary to complete the proof. As future work, we plan to simulate this process in Athena by translating back from Athena to CafeOBJ in order to help the formulation of intermediate lemmas. Our approach could be used as a vehicle for integrating other algebraic specification methods (such as Maude and CASL) with more conventional theorem-proving systems based on first- or higher-order logic.

Acknowledgments This research has been co-financed by the European Union (European Social Fund ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: THALIS

References

1. Cafeobj. http://www.ldl.jaist.ac.jp/cafeobj
2. Maude. http://maude.cs.uiuc.edu
3. Casl. http://www.casl.umd.edu/about
4. Hol/isabelle. http://isabelle.in.tum.de/library/HOL
5. Diaconescu, R., Futatsugi, K.: CafeOBJ report: The language, proof techniques, and methodologies for object-oriented algebraic specification. AMAST Series in computing **6** (1998)
6. Diaconescu, R., Futatsugi, K., Ogata, K.: CafeOBJ: Logical foundations and methodologies. Computers and Artificial Intelligence **22**(3-4) (2003) 257–283
7. Arkoudas, K.: Athena. http://proofcentral.org/athena/ (2004)
8. Futatsugi, K., Găină, D., Ogata, K.: Principles of proof scores in CafeOBJ. Theoretical Computer Science **464** (2012) 90–112
9. Găină, D., Lucanu, D., Ogata, K., Futatsugi, K.: On automation of OTS/CafeOBJ method. In Iida, S., Meseguer, J., Ogata, K., eds.: Specification, Algebra, and Software. Volume 8373 of LNCS. Springer (2014) 578–602

Symbolic Execution by Language Transformation

Andrei Arusoaie[1], Dorel Lucanu[1], and Vlad Rusu[2]

[1] Faculty of Computer Science, Alexandru Ioan Cuza University, Iaşi, Romania
andrei.arusoaie@gmail.com,dlucanu@info.uaic.ro
[2] Inria Lille Nord Europe, France vlad.rusu@inria.fr

In [1] we presented a language-independent symbolic execution framework based on the formal operational semantics of programming languages, together with an implementation of our approach in the $\mathbb{K}$ language-definition framework [2]. Briefly, the approach in [1] consisted in transforming a language $\mathcal{L}$ into a language $\mathcal{L}^{\mathfrak{s}}$, so that symbolic execution of a program in $\mathcal{L}$ is, by definition, the standard execution of the same program seen as an element of $\mathcal{L}^{\mathfrak{s}}$. We proved that the resulting notion of symbolic execution has the properties naturally expected of it (mutual simulation with concrete execution).

In this paper we present symbolic execution under a different angle, and investigate the relationship between this notion of symbolic execution and the one based on the $\mathcal{L}^{\mathfrak{s}}$ construction of [1]. The goal is to be able to implement the new notion of symbolic execution (based on *symbolic rewriting*) in a setting where only standard rewriting is available, such as the $\mathbb{K}$ framework with its Maude backend, and to be able to do this for real-size programming languages.

Programming languages are tuples of the form $\mathcal{L} \triangleq ((\Sigma, \Pi), \mathit{Cfg}, \mathcal{T}, \mathcal{S})$ where (Σ, Π) is a first-order signature, *Cfg* a distinguished sort for *configurations* (which consists of code together with whatever information is required for executing it - depending on the language, e.g., variable store, heap, stack; ...); $\mathcal{T}$ is a (Σ, Π)-model; and $\mathcal{S}$ is a set of *reachability rules* of the form $\varphi \Rightarrow \varphi'$ defining the operational semantics of $\mathcal{L}$. For each language, the many-sorted signature Σ contains a subsignature $\Sigma^{\mathfrak{d}}$ which includes all the *data* sorts of the language (e.g. integers, booleans, ... again, depending on the language being defined).

Our approach uses unification for formally defining symbolic execution. A *symbolic unifier* of two terms t_1, t_2 is any substitution $\sigma : \mathit{var}(t_1) \cup \mathit{var}(t_2) \to T_\Sigma(Z)$ for some set Z of variables such that $\sigma(t_1) = \sigma(t_2)$. A *concrete unifier* of terms t_1, t_2 is any valuation $\rho : \mathit{var}(t_1) \cup \mathit{var}(t_2) \to \mathcal{T}$ such that $\rho(t_1) = \rho(t_2)$. A symbolic unifier σ of two terms t_1, t_2 is a *most general unifier of* t_1, t_2 with respect to concrete unification whenever, for all concrete unifiers ρ of t_1 and t_2, there is a concrete unifier $\eta : Z \to \mathcal{T}$ such that $\eta \circ \sigma = \rho$. We prove that under some conditions on the semantics of $\mathcal{S}$ (which can always be obtained, provided that the only symbolic values are data) a unique most-general unifier exists.

Symbolic execution essentially consists of applying the semantical rules over patterns using most general unifiers. Two patterns φ and φ' are in relation $\sim$, i.e. $\varphi \sim \varphi'$, iff they match the same set of configurations (we write $[\![\varphi]\!] = [\![\varphi']\!]$); a configuration γ *matches* against a pattern $\varphi \triangleq \pi \wedge \phi$ if there is a valuation $\rho : \mathit{Var} \to \mathcal{T}$ such that $\rho(\pi) = \gamma$ and $\rho \models \phi$, i.e., for a certain valuation of the variables occurring in the pattern, its condition ϕ is satisfied and the

configuration γ is an instance of the *basic* pattern π. $\sim$ is an equivalence relation (we let $[\varphi]_\sim$ denote the equivalence class of φ). We define the symbolic transition relation $\Rightarrow^{\mathfrak{s}}_{\mathcal{S}}$ by: $[\varphi]_\sim \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} [\varphi']_\sim$ iff $\varphi \triangleq \pi \wedge \phi$, all the variables in $var(\pi)$ have data sorts, there is a rule $\alpha \triangleq \varphi_1 \Rightarrow \varphi_2 \in \mathcal{S}$ with $\varphi_i \triangleq \pi_i \wedge \phi_i$ for $i \in \{1, 2\}$, $var(\varphi) \cap var(\alpha) = \emptyset$, π_1 and π are concretely unifiable, and $[\varphi']_\sim = [\sigma^{\pi_1}_{\pi}(\pi_2) \wedge \sigma^{\pi_1}_{\pi}(\phi \wedge \phi_1 \wedge \phi_2)]_\sim$, where $\sigma^{\pi_1}_{\pi}$ is the most general symbolic unifier of π, π_1., extended as the identity substitution over the variables in $var(\phi_1, \varphi_2) \backslash var(\pi, \pi_1)$.

With this definition of symbolic execution we proved that there is a natural relation between symbolic and concrete program execution:

- to every concrete execution there corresponds a feasible symbolic one;
- to every feasible symbolic execution there corresponds a concrete one;

where to executions *correspond* if they follow the same program path.

However, our definition of symbolic execution requires rules to be applied in a *symbolic* manner, i.e., using symbolic unifiers as shown above.

A question that arises is then: how to implement this symbolic rewriting in a setting where only standard rewriting is available, such the $\mathbb{K}$ language-definition framework? The answer is to adapt the approach from [1]: to transform $\mathcal{L}$ into a new language definition $\mathcal{L}^{\mathfrak{s}}$, and to ensure that execution by rewriting in $\mathcal{L}^{\mathfrak{s}}$ coincides with execution by symbolic rewriting in $\mathcal{L}$. In [1] we considered a $\Sigma^{\mathfrak{d}}$-model $\mathcal{D}$ which interprets the data sorts and operations, and a free Σ-model $\mathcal{T}$ generated by $\mathcal{D}$ which interprets the non-data sorts as ground terms over the signature $(\Sigma \setminus \Sigma^{\mathfrak{d}}) \cup \bigcup_{d \in Data} \mathcal{D}_d$ (the elements of the carrier sets $\mathcal{D}_d$ are added as new constants in the signature). The components of $\mathcal{L}^{\mathfrak{s}}$ are automatically obtained from $\mathcal{L}$. In particular, the symbolic model $\mathcal{T}^{\mathfrak{s}}$ is freely generated by $\mathcal{D}^{\mathfrak{s}}$, where $\mathcal{D}^{\mathfrak{s}}$ is $\mathcal{D}$ enriched with *symbolic values* from the symbolic signature $\Sigma^{\mathfrak{s}}$.

In this paper we consider a more general class of models $\mathcal{T}$, which is rich and expressive enough for defining real-size languages. Our goal is to give an axiomatic definition of $\mathcal{T}$ and a procedure to built $\mathcal{T}^{\mathfrak{s}}$ from $\mathcal{T}$. The axiomatic definition for $\mathcal{T}$ captures the essence of what the model is, without constraining it too much (e.g., not requiring it to be freely generated). We are also interested in establishing the equality between $\Rightarrow^{\mathfrak{s}}_{\mathcal{S}}$ (the symbolic transition relation of $\mathcal{L}$, defined above) and $\Rightarrow_{\mathcal{S}^{\mathfrak{s}}}$, which is the transition relation of the symbolic language $\mathcal{L}^{\mathfrak{s}}$. This would ensure that (our new notion of) symbolic execution is still well-behaved (i.e., in mutual simulation) with respect to concrete execution.

References

1. Andrei Arusoaie, Dorel Lucanu, and Vlad Rusu. A generic framework for symbolic execution. In *6th International Conference on Software Language Engineering*, volume 8225 of *LNCS*, pages 281–301. Springer Verlag, 2013. Also available as a technical report `http://hal.inria.fr/hal-00853588`.
2. Traian Florin Serbanuta, Andrei Arusoaie, David Lazar, Chucky Ellison, Dorel Lucanu, and Grigore Rosu. The k primer (version 3.3). In *Proceedings of International K Workshop (K'11)*, ENTCS. Elsevier, 2013. To appear.

Programming Language Abstractions Based on Executable Algebraic Semantics

Irina Măriuca Asăvoae

VERIMAG/Universitè Joseph Fourier, France
irina.asavoae@imag.fr

The program analysis and verification methods are generally characterized by the infamous EXPTIME complexity. This generates an effectiveness problem for the program analysis/verification, problem which is traditionally by-passed by the abstract interpretation [5,4]. Namely, the abstract interpretation framework standardizes a method for reducing the complexity of the analysis/verification methods by a *coherent projection* of the program's state space. The projection is actually a mapping from the state space of the *concrete* system into the state space of an *abstract* system such that the mapping preserves the transition relations. By coherent projection we understand that the class of properties currently analyzed/verified is reflected from the abstract system into the concrete system. Furthermore, sharp abstractions may also require an associated projection of the concrete transition operators into the abstract ones [3]. The coherent projection is founded by a Galois connection between the concrete and the abstract systems which are, in turns, viewed as lattices. The abstract interpretation framework is developed on top of the operational semantics and there exists an abundance of literature on practical instantiations of it [9].

On the other hand, the rewriting logic already has established its own notion of abstraction, namely the algebraic simulation [8,7]. In more details, the standard notion of simulation is incrementally mirrored into the rewriting logic under the hypothesis that the systems being related are described as rewriting systems. The presentation in [8,7] is eloquently sustained at each step by examples instantiating the concrete and the abstract rewrite systems and by providing the definition of the algebraic simulation also as a rewrite system. The beauty of this approach lies in the unified rewriting logic representation of both the systems and the simulation. Moreover, the approach is automatically executable using the tools implementing the rewriting logic [2]. Consequently, the rewriting logic and the algebraic simulations provide an appealing framework for specifying abstractions using the algebraic denotational semantics.

The view of the current work is drawn from the above stated facts. One fact is that Rewriting Logic Semantics unifies the operational semantics and the algebraic denotational semantics. Another fact is that the abstractions already proved to be a sine qua non in the field of analysis and verification. These two facts induce the natural idea of a systematic transportation of the results from the abstract interpretation into the Rewriting Logic Semantics. As such, the current work studies the transformation of Galois connections into theoroidal maps with a direct application to a transformation from LLVM intermediate representation (LLVM–IR) [1] into MIR—a language with reduced set of instructions which is used for program analysis. Namely, the concrete programming language, LLVM–IR, is first *syntactically mapped* into the abstract programming language, MIR. Then, we define a projection of the *syntactic mapping* at

the level of the rewriting rules defining the semantics of these two languages. Next, we study to what extend the syntactic mapping of the two languages preserves the memory model of the two languages. This last part allows us to to argue to what extend we can say that the properties verified on MIR are verifyaing properties of LLVM–IR. Finally, we adjust the proposed transformation to the stages of abstraction described in [11], and we exemplify using the notorious predicate abstraction on the abstract language. The choice of predicate abstraction is justified by the existence of consistent documentations in both worlds, namely in abstract interpretation [6] and in rewriting logic [10].

References

1. LLVM language reference manual. http://llvm.org/docs/LangRef.html
2. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L. (eds.): All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic, Lecture Notes in Computer Science, vol. 4350. Springer (2007)
3. Cousot, P.: Design of syntactic program transformations by abstract interpretation of semantic transformations (invited talk). In: Codognet, P. (ed.) Proceedings of the Seventeenth International Conference, ICLP 2001. pp. 4–5. LNCS 2237, Springer, Berlin, Paphos, Cyprus (November/December 2001)
4. Cousot, P.: Formal verification by abstract interpretation. In: Goodloe, A., Person, S. (eds.) 4th NASA Formal Methods Symposium (NFM 2012). Lecture Notes in Computer Science, vol. 7226, pp. 3–7. Springer-Verlag, Heidelberg (2012)
5. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Symposium on Principles of Programming Languages. pp. 238–252. ACM Press (1977)
6. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Proceedings of the 9th Conference on Computer-Aided Verification. pp. 72–83. Springer-Verlag (1997)
7. Meseguer, J., Palomino, M., Martí-Oliet, N.: Equational abstractions. Theor. Comput. Sci. 403(2-3), 239–264 (2008)
8. Meseguer, J., Palomino, M., Martí-Oliet, N.: Algebraic simulations. J. Log. Algebr. Program. 79(2), 103–143 (2010)
9. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag New York, Inc. (1999)
10. Palomino, M.: A predicate abstraction tool for Maude. Documentation and tool available at `http://maude.sip.ucm.es/~miguelpt/bibliography.html`
11. Schmidt, D.A., Steffen, B.: Program analysis *as* model checking of abstract interpretations. In: Levi, G. (ed.) Static Analysis, 5th International Symposium, SAS '98, Pisa, Italy, September 14-16, 1998, Proceedings. Lecture Notes in Computer Science, vol. 1503, pp. 351–380. Springer (1998)

Semantic Mining of Context Update Constructs in Imperative Languages

Irina Măriuca Asăvoae[1], Mihail Asăvoae[1], Adrián Riesco[2]

[1] VERIMAG/Université Joseph Fourier, France
{irina.asavoae, mihail.asavoae}@imag.fr
[2] Universidad Complutense de Madrid, Spain
ariesco@fdi.ucm.es

In the view of an easier development, complex software systems are constructed in a modular fashion. Modularity has various structural representations which vary according to the programming paradigm where it appears. For example, the modularity in imperative languages is given by functions and procedures, in object-oriented languages by classes and interfaces, while in declarative programming by modules. Besides its structural characteristic, the modularity carries functional information which sets the relation between the modularity units. For example, in imperative languages the relation between modules is given by function calls, in object-oriented languages by the class inheritance, while in declarative programming we find a similar module hierarchy. During the development of the software systems both structural and functional characteristics of the modularity are heavily exploited to shape the final product.

An important part of the software development is the system evaluation stage, which includes testing, analysis, and verification. Due to the increased system complexity, it is advisable to interleave the system construction with its evaluation. Moreover, the theoretical results for evaluation methods show that an efficient evaluation should be designed in a structural inductive fashion, by focusing first on the unit evaluation and then moving to the compositional evaluation. Consequently, the complexity of the systems corroborated with the efficiency requirement of their evaluation methods induce the necessity of preserving and exploiting the modular characteristics of the system during its evaluation.

Rewriting logic [3] proposes an algebraic and executable setting to integrate the construction and the evaluation of systems. In the current work we consider these systems as programs written in some programming language. The integration methodology proposed by rewriting logic starts with $\mathcal{S}$—a formal executable semantics of the programming language used for the system development. $\mathcal{S}$ provides the set of all *concrete executions*, for any program correctly constructed w.r.t. the language syntax and for all possible input data. Here we consider $\mathcal{S}$ to be from the family of imperative languages and we focus on inferring the modularity aspects in imperative languages. To achieve this, we apply *semantic mining*, i.e., we analyze $\mathcal{S}$ in order to extract the programming language constructs which induce the functional aspects of the modularity. We define $\mathcal{S}$ as a rewriting logic theory [3] which is executable and benefits of tool support via the Maude system [1], an implementation of the rewriting logic framework.

We presented in [4] a methodology for performing generic intraprocedural slicing, while [5] improved it by presenting a generic interprocedural slicing. However, the generality of the interprocedural slicing depends on the automatic inference of the modularity attributes of the language in study. Recall that for imperative languages we identify as modularity features the function calls through which each function activates its own context. We call *context-updates* the language syntactic constructs that denote the function calls. Note that, modulo variations of parameter passing, the context-updates induce the functional aspect of the modularity in imperative languages and we show in [5] that they are a key element in generic interprocedural slicing.

This paper extends our work on generic slicing by showing how to extract context-update constructs $\mathfrak{c}$ from the given language semantics, $\mathcal{S}$, and how to use them as input for the interprocedural slicing for $\mathcal{S}$-programs described in [5]. Our method for obtaining the context-update language constructs is, in fact, a homomorphism $\mathfrak{H}$ which takes the algebra defined by $\mathcal{S}$ and maps it into an algebra which has as operations only the language constructs $\mathfrak{c}$.

For the construction of $\mathfrak{H}$, we consider as given both $\mathcal{S}$, the sort(s) in $\mathcal{S}$ for the language constructs *con*, and the sort(s) defining the memory of the program *mem*. The construction of $\mathfrak{H}$ explores the rewrite rules in $\mathcal{S}$, starting with the rules which rewrite terms of sort *con*, in search of a subsequent set of rules which builds a stack of *mem* terms. The results of $\mathfrak{H}$ are the *context-updates*. We exemplify the method on a particular $\mathcal{S}$—an extension of the WHILE language [2] in which we introduce variable scoping, i.e., homonymous variables behave differently w.r.t. the context where they are declared and used. We call this extension WhileF which is a simple imperative language with assignments, conditions, and loops, enriched with constructs like functions and local variables.

A more refined feature of context-updates is the parameter passing and we aim to further study the context-updates towards this particular aspect.

References

1. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude: A High-Performance Logical Framework, Lecture Notes in Computer Science, vol. 4350. Springer (2007)
2. Hennessy, M.: The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics. John Wiley & Sons (1990)
3. Martí-Oliet, N., Meseguer, J.: Rewriting logic: roadmap and bibliography. Theor. Comput. Sci. 285(2), 121–154 (2002)
4. Riesco, A., Asavoae, I.M., Asavoae, M.: A generic program slicing technique based on language definitions. In: Martí-Oliet, N., Palomino, M. (eds.) Proceedings of the 21st International Workshop on Algebraic Development Techniques, WADT 2012. Lecture Notes in Computer Science, vol. 7841, pp. 248–264. IFIP International Federation for Information Processing (2013)
5. Riesco, A., Asavoae, I.M., Asavoae, M.: A generic program slicing technique based on language definitions. In: Preproceedings of the 20th Workshop of Rewriting Logic and Its Applications, WRLA 2014. http://users.dsic.upv.es/workshops/wrla2014/#proceedings (2014)

An Institutional Approach to Positive Coalgebraic Logic

Adriana Balan[1], Alexander Kurz[2], and Jiří Velebil[3] *

[1] University Politehnica of Bucharest, Romania
adriana.balan@mathem.pub.ro
[2] University of Leicester, UK
ak155@mcs.le.ac.uk
[3] Faculty of Electrical Engineering, Czech Technical University in Prague, Czech Republic
velebil@math.feld.cvut.cz

Institutions provide a general framework for describing logical systems; in particular, such is the case for the coalgebraic modal logic, built upon the observation that (finitary) *predicate liftings* generalize modal operators from Kripke frames to coalgebras over arbitrary Set-functors. It was shown in [3,6] that this specific logic carries the structure of an institution, with Set-endofunctors as signatures and (opposites of) natural transformations between them as morphisms of signatures, models being the associated categories of coalgebras.

Coalgebraic modal logic has its roots in the modal logic of Kripke frames. The latter is given by atomic propositions, Boolean operations, and a unary box preserving finite meets. In [4], Dunn gave an axiomatisation of the positive fragment of this logic, by atomic propositions, *lattice* operations, and unary box and diamond, but no negation.

In [1] and in the subsequently expanded version [2], we have generalized his results from Kripke frames to coalgebras for arbitrary functors preserving weak pullbacks. More in detail, an arbitrary functor $T : \mathsf{Set} \to \mathsf{Set}$ induces by duality a (finitary) functor $L : \mathsf{BoolAlg} \to \mathsf{BoolAlg}$ on the category of Boolean algebras (the categorical counterpart of the corresponding modal logic), so that the free L-algebras are exactly the Lindenbaum algebras of the modal logic. Following Dunn's ideas, we use to the category DLat of (bounded) distributive lattices as a base for the positive fragment of (the logic corresponding to) L, that is, for a (strongly finitary) functor $L' : \mathsf{DLat} \to \mathsf{DLat}$ which, in turn, by duality, arises from a (locally monotone) functor $T' : \mathsf{Poset} \to \mathsf{Poset}$ on the category Poset of posets and monotone maps. An important point is that we have to work in a Poset-enriched setting, to guarantee that the modal operations (like $\Box$ and $\Diamond$ in positive modal logic) are monotone, and that the logic L' admits a monotone presentation.

To obtain an institutionalized approach of the results in [1,2], we had to consider a Poset-enrilched version of the notion of institution, with models lying in Poset-Cat and sentences in Poset. The motivating example, that we shall denote by Ins', has locally monotone functors on Poset as signatures and their categories

* Supported by the grant no. P202/11/1632 of the Czech Science Foundation.

of coalgebras as models. Examples of models (coalgebras for locally monotone functors) include ordered automata (automata with a partial order of states and monotone transitions), or frames for distributive substructural logics. To fit within the framework of Poset-enrichment, the satisfaction relation between models and sentences has to be monotone itself, where models (coalgebras) are ordered through simulations. This requires that the signature functors must have the Beck-Chevalley property (the Poset-analogue of preserving weak pullbacks property).

The institution Ins mentioned in the first paragraph, from [3,6], can then be adapted to our Poset-enriched setting, by considering Set and $\mathsf{BoolAlg}$ discretely Poset-enriched.

Then the institutions of coalgebraic modal logic Ins (restricted to signature functors that preserve weak pullbacks) and Ins' are related by a (Poset-enriched) morphism of institutions $\mathsf{Ins} \to \mathsf{Ins}'$, which sends $T : \mathsf{Set} \to \mathsf{Set}$ to its canonical extension $T' : \mathsf{Poset} \to \mathsf{Poset}$ (called *posetification* in [1,2]), realized as a completion with respect to Poset-enriched colimits. The construction $T \mapsto T' \mapsto L'$ gives the maximal positive fragment of L (thus generalizes Dunn's theorem), and allows to define the morphism of institutions on the level of sentences.

References

1. Balan, A., Kurz, A., Velebil, J.: *Positive Fragments of Coalgebraic Logics.* In: CALCO2013, LNCS 8089, 51–65 (2013)
2. Balan, A., Kurz, A., Velebil, J. *Positive Fragments of Coalgebraic Logics*, extended version, submitted, available at arXiv:1402.5922, 38 pp. (2014)
3. Cîrstea, C. *An institution of modal logics for coalgebras*, J. Logic Alg. Progr. 67(1-2), 87–113 (2006)
4. Dunn, J. M.: *Positive Modal Logic.* Studia Logica 55, 301–317 (1995)
5. Goguen, J., R. Burstall. *Institutions: Abstract Model Theory for Specification and Programming.* J. Assoc. Comput. Mach. 39(1), 95–146 (1995)
6. Pattinson, D. *Translating logics for coalgebras.* In: WADT2002, LNCS 2755, 393–408 (2003)
7. Worrell, J. *Coinduction for recursive data types: partial order, metric spaces and Ω-categories*, Electron. Notes Theor. Comput. Sci. 33, 367–386 (2000)

Autonomous Systems in Rewriting Logic ⋆

Lenz Belzner

LMU Munich, PST Chair
belzner@pst.ifi.lmu.de

Autonomous systems are on the rise, and modern software systems more and more are enabled to take decisions at runtime autonomously in order to react adequately to unforeseen situations and to reach system goals in the face of uncertainty. While these abilities are appealing, their incorporation into a system gives rise to new challenges: System specification has to incorporate knowledge about the domain, as well as algorithms that allows an agent to interpret its knowledge in a particular situation in order to derive goal based behaviour. Also, verification of certain behavioural properties of systems that act and take decisions autonomously before actually executing them may be a valuable consideration. Thus, approaches to specification for autonomous systems should provide features that allow to perform verification tasks.

Relational Markov decision processes (RMDPs) [1,2] serve as a formal framework for *sequential decision making under uncertainty*, and previous work of the author introduced the encoding of RMDPs in rewriting logic [3]. An action programming language has been introduced that allows system designers to specify behavioural constraints for agents [3,4]. Specified agent behaviour is evaluated at runtime by term rewriting according to system specification, taking into account system state, domain non-determinism and specified system goals to allow for autonomous decision making. Offline computation of optimal policies that are maximizing reward at runtime can be achieved by performing symbolic value iteration for rewrite theories [5].

The rewriting logic approach to symbolic artificial intelligence differs in three main points from alternative approaches as for example the situation calculus [6]. First, specification and reasoning is possible for *arbitrary* first-order algebraic domain representations. Thus, the representational distance between problem domain and actual encoding is minimized, easing specification and communication about it [7]. Second, specification of domain dynamics as rewrite rules offers an intuitive solution to the notorious frame problem [6], alleviating specification complexity by avoiding the need to explicitly specify system properties that remain unchanged by action execution. Finally, rewrite theory specifications are executable in the Maude language (given they satisfy certain admissibility requirements), which offers a rich toolset for system verification [8].

The talk will expand on two current research directions in the field. First, confluence, coherence and termination of the language specification are proved, supported by the Maude CRChC tool [9]. This result provides a formal underpinning to the specification approach, and ensures executability of specifications in the Maude language.

⋆ This work has been partially funded by the EU project ASCENS, 257414.

Second, while previous work provided answers to the question "*What should an agent do (provided it knows everything about its environment)?*", current work copes with the question "*What should an agent know?*". Typically, an agent's knowledge about its environment is limited, and acquisition of knowledge, e.g. through sensing or communication, is a costly task in terms of system resources as e.g. energy, memory or bandwidth. Still, there may be valuable information available that would provide justification to some goal based behaviour.

In order to decide on what information to gather, it is argued that replacing rewriting with narrowing when interpreting an action program in a setting with limited knowledge can optimize an agent's attempts to gather knowledge w.r.t. system specification. The main idea is that instead of *matching* current information about the environment with general knowledge about action consequences when performing rewriting, these are *unified* when performing narrowing. Unification provides a set of *hypotheses* to reason about and evaluate, serving as indicators about the value of acquiring particular information. Possible knowledge acquisition actions (e.g. active sensing or requesting information through communication) are evaluated w.r.t. their cost and possible future reward, allowing an agent to autonomously decide what information to gather.

References

1. Bellman, R.: A markovian decision process. Indiana Univ. Math. J. **6** (1957) 679–684
2. Sanner, S., Boutilier, C.: Approximate linear programming for first-order MDPs. In: UAI, AUAI Press (2005) 509–517
3. Belzner, L.: Verifiable decisions in autonomous concurrent systems. In Kühn, E., Pugliese, R., eds.: COORDINATION. Volume 8459 of Lecture Notes in Computer Science., Springer (2014) 17–32
4. Belzner, L.: Action programming in rewriting logic. TPLP **13**(4-5-Online-Supplement) (2013)
5. Belzner, L.: Value iteration for relational MDPs in rewriting logic. In Endriss, U., Leite, J., eds.: STAIRS. Frontiers in Artificial Intelligence and Applications, IOS Press (2014) to appear
6. Reiter, R.: Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems. Illustrated edn. The MIT Press, Massachusetts, MA (2001)
7. Meseguer, J.: Twenty years of rewriting logic. J. Log. Algebr. Program. **81**(7-8) (2012) 721–781
8. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L., eds.: All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic. Volume 4350 of Lecture Notes in Computer Science., Springer (2007)
9. Durán, F., Meseguer, J.: A church-rosser checker tool for conditional order-sorted equational maude specifications. [10] 69–85
10. Ölveczky, P.C., ed.: Rewriting Logic and Its Applications - 8th International Workshop, WRLA 2010, Held as a Satellite Event of ETAPS 2010, Paphos, Cyprus, March 20-21, 2010, Revised Selected Papers. In Ölveczky, P.C., ed.: WRLA. Volume 6381 of Lecture Notes in Computer Science., Springer (2010)

Arguing Safety Cases Formally

Valentín Cassano[1] and Thomas S.E. Maibaum[1]

Department of Computing and Software, McMaster University
cassanv@mcmaster.ca tom@maibaum.org

Abstract

In the certification of safety critical systems, *evidence-based* safety regimes are starting to gain prominence over the more traditional *process-based* ones. For instance, evidence-based safety regimes are being put in place in order to demonstrate compliance with automotive safety standards, with medical device regulations, and with commercial airplane software-based systems safety requirements (q.v. [1], [2], [3], respectively).

In brief, whereas a process-based safety regime typically requires adherence to a particular set of prescriptions which must be followed in order to assure that a system is safe for its intended use, an evidence-based safety regime typically requires the production of what is known as a *safety case*. In essence, a safety case is intended to make a compelling case that adequate levels of safety have been achieved. To this end, a safety case is seen as being comprised of: an explicit set of safety goals which must be met, evidence that these goals are actually met, and a structured argument linking evidence to safety goals. There are two main assumptions underlying this notion of a safety case. The first of these asserts that: "assuring that a system is safe without adequate evidence of this being the case is, or better yet, should be, unconvincing". The second of these asserts that: "in the absence of a structured argument linking evidence to safety goals, evidence is left unexplained; hence, a shadow of doubt is casted over whether safety goals have actually been met".

However, in spite of some significant benefits, evidence-based safety regimes are not a panacea. What can be seen as their greatest virtue, the notion of a safety case, is also their 'Achilles' heel'. Issues such as completeness of safety goals, scientific rigour and sufficiency of evidence, and validity of safety arguments, i.e., those linking evidence to safety goals, are seldom adequately addressed. In this work, we will focus on studying safety arguments from a formal standpoint, so that their validity can be assessed in a rigorous manner.

Of course, this formal study would be far more straightforward if safety arguments were to resort to the typical elements of deductive logic in discussions related to the way in which evidence supports the claim that safety goals are actually met. But this faint hope exposes its hollowness after a quick review of some typical safety cases (q.v. [4]). It is common to encounter in them judgments of experts, hasty inductive generalizations, and fallacious uses of language, all things which are far from being logical. Complicating this picture, the safety community considers that only in simple cases do safety arguments agree with

what traditional logical studies view as their formal counterpart. More precisely, in traditional logical studies, arguments are formalized as proofs, with proofs being defined with respect to a judiciously chosen set of rules of inference. These rules of inference are usually those of classical logic or variants such as those dealing with modalities. However, statements such as "data such as D, *normally* entitles one to make a claim such as C", found everywhere in safety arguments, do not seem to correspond well with the type of statements made by traditional rules of inference. Instead, we view them as being better captured by those argument patterns discussed by Toulmin in [5]. In that work, Toulmin advances that argument patterns which are found in everyday reasoning are usually comprised of *claims* that are established from *data* resorting to, so called, *warrants*. Warrants are statements acting as 'bridges' between claims and the data which support them, such as the one mentioned above. Toulmin then goes on and discusses that while "some warrants authorize us to accept a claim unequivocally, given the appropriate data [...] others authorize us to make the step from data [...] either tentatively, or else subject to conditions, exceptions, or qualifications". Whereas the first kind of warrant can be seen as agreeing with the traditional view of a rule of inference, warrants of the second kind do not, and are better understood as being defeasible.

Aiming at presenting a context in which safety arguments could be studied formally, in this work we will discuss in which way Toulmin's defeasible argument patterns can be seen as being captured by *default rules* (q.v. [6]). In doing this, we resort to a proposed new interpretation of default rules: we view them as indicating claims which are held tentatively. We will comment on how this formalization reflects some observations we have made while studying safety arguments. In turn, potential contributions of our approach include a formal assessment of: the internal coherence and validity of safety arguments, the relevance of those premises acting in support of a claim, and the strength of a conclusion a safety argument aims at establishing. Moreover, we contend that, since 'bridge' steps between data and the claims they support are made explicit, our approach eases the analysis of fallacies in safety arguments; resulting in them being less questionable from a logical standpoint and increasing the overall confidence that is placed on the safety of a system. Lastly, we consider that our approach enables for 'safety argument patterns' to be elicited from safety arguments, with the obvious benefits this entails.

References

1. Int. Standard Organization: ISO 26262: Road Vehicles – Functional Safety (2011)
2. Food and Drug Administration: 510(k) Clearance and Premarket Approval (2014)
3. RTCA SC-205, EUROCAE WG-12: DO-178C: Software considerations in airborne systems and equipment certification (2012)
4. Dependability and Security Research Group at UVa: Safety cases repository (2006)
5. Toulmin, S.E.: The Uses of Argument. Cambridge University Press (2003)
6. Reiter, R.: A logic for default reasoning. Artificial Intelligence **13**(1-2) (1980) 81–132

Proving Properties of Concurrent Systems using Graph Transformations and Event-B*

Simone A. C. Cavalheiro[1], Luciana Foss[1], and Leila Ribeiro[2]

[1] {simone.costa,lfoss}@inf.ufpel.edu.br
Federal University of Pelotas (UFPEL), Pelotas, Brazil
[2] leila@inf.ufrgs.br
Federal University of Rio Grande do Sul (UFRGS), Porto Alegre, Brazil

Graph transformation (GT) is a formal specification language well-suited to applications in which states have a complex topology and in which behavior is essentially data-driven, that is, events are triggered basically by particular configurations of the state [5]. Reactive systems, like protocols for distributed and mobile systems, and model-driven transformations are examples of this class of applications. GT provides a formal yet intuitive (and visual) specification language, offering a rich notion of states (represented as graphs) and a simple but powerful notion of state change (defined by rules). In addition, in the GT approach, concurrency and non-determinism appear naturally: many events (rule applications) may happen concurrently, if they all are enabled, and the choice of occurrence between conflicting events is non-deterministic.

To allow the specification of practical applications, two extensions of the basic GT formalism are particularly relevant: Negative application conditions (NACs) [8] and attributes[6]. NACs allow the specification of situations that shall prevent rule application, in addition to the situations which allow rule application. The use of this feature typically reduces the number of rules required to specify a system, and also results in clearer rules. The use of elements of data types as attributes of vertices and edges, and also of variable and equations in rules, also helps in making GT specifications more intuitive (and smaller).

There is a variety of graphical tools for specifying, executing and analysing GTs (see [7] for a comparison of some of them). There are syntactic analysis techniques, like critical pairs and concurrent rules, and they can be performed by tools like AGG even in the presence of NACs and attributes. Model checking approaches in GT (like GROOVE [9]), however, handle NACs and attributes in a limited way, since the construction of the state-space in the case of GTs may lead to huge structures, even in the case without attributes.

Besides model checking, another well-established approach used for verification purposes is theorem proving. In this technique, both the system and its desired properties are expressed as formulas in some mathematical logic. Theorem proving can deal directly with infinite state spaces and it relies on techniques such as structural induction to prove properties over infinite domains. In [3,4] we introduced an approach to GTs, providing an encoding of graphs and rules into relations, that enabled the use of logic formulas to express properties of

* This work is part of the VeriTeS project, supported by FAPERGS and CNPq

reachable states and allowed the use of Event-B [1] theorem provers to analyze properties of GTs [2,10]. In this paper we extend the approach to deal with GTs with NACs and attributes. In [4] we defined the theoretical foundations of the logical description of GTs with attributes, which is the basis for the work presented here. Our main contribution is a technique to allow the analysis of infinite-state GTs with NACs and attributes by using theorem proving. The approach is based on a translation of GTs with NACs and attributes to event-B. Our proposal complements the existing approaches for GTs analysis.

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, New York, NY, USA, 1st edn. (2010)
2. da Costa, S.A.: Relational Approach of Graph Grammars. Ph.D. thesis, INF, UFRGS, Brazil (2010)
3. da Costa, S.A., Ribeiro, L.: Formal verification of graph grammars using mathematical induction. ENTCS 240, 43–60 (2009)
4. da Costa, S.A., Ribeiro, L.: Verification of graph grammars using a logical approach. Science of Computer Programming 77(4), 480 – 504 (2012)
5. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G. (eds.): Handbook of graph grammars and computing by graph transformation: vol. 2: applications, languages, and tools. World Scientific, River Edge, USA (1999)
6. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamental theory for typed attributed graphs and graph transformation based on adhesive hlr categories. Fundamta Informaticae 74(1), 31–61 (2006)
7. Fuss, C., Mosler, C., Ranger, U., Schultchen, E.: The jury is still out: A comparison of Agg, Fujaba, and PROGRES. ECEASST 6 (2007)
8. Habel, A., Heckel, R., Taentzer, G.: Graph grammars with negative application conditions. Fundamenta Informaticae 26(3-4), 287–313 (1996)
9. Rensink, A.: The GROOVE simulator: A tool for state space generation. In: Pfalz, J., Nagl, M., Böhlen, B. (eds.) Applications of Graph Transformations with Industrial Relevance (AGTIVE). LNCS, vol. 3062, pp. 479–485. Springer (2004)
10. Ribeiro, L., Dotti, F.L., da Costa, S.A., Dillenburg, F.C.: Towards theorem proving graph grammars using event-b. ECEASST issue on GraMoT 2010 30 (2010)

An Institutional Foundation for the $\mathbb{K}$ Semantic Framework

Claudia Elena Chiriţă and Traian Florin Şerbănuţă

Faculty of Mathematics and Computer Science, University of Bucharest
claudia.elena.chirita@gmail.com, traian.serbanuta@gmail.com

The $\mathbb{K}$ framework is an executable semantic framework based on rewriting and used for defining programming languages, computational calculi, type systems and formal-analysis tools. It was developed as an alternative to the existing operational-semantics frameworks and over the years has been employed to define actual programming languages, to study runtime verification methods and to develop analysis tools such as type checkers, type inferencers, model checkers and verifiers based on Hoare-style assertions. A comprehensive overview of the framework can be found in [1]. Its associated tool [2] enables the development of modular and executable definitions of languages, and moreover, it allows the user to test programs and to explore their behaviour in an exhaustive manner, facilitating in this way the design of new languages.

Driven by recent developments on the theoretical foundations of the $\mathbb{K}$ semantic framework [3,4] and on the established connections with other semantic frameworks and formal systems such as reduction semantics, Hoare logic and separation logic, we propose an institutional formalisation [5] of the logical systems on which the $\mathbb{K}$ framework is based: matching and reachability logic. This would allow us to extend the usage of $\mathbb{K}$ by focusing on its potential as a formal specification language, and furthermore, through its underlying logics, to establish rigorous mathematical relationships between $\mathbb{K}$ and other similar languages, enabling the integration of their verification tools and techniques.

Matching logic [3] is a formal system used to express properties about the structure of mathematical objects and language constructs, and to reason about them by means of pattern matching. Its sentences, called patterns, are built in an inductive manner, similarly to the terms of first-order logic, using operation symbols provided by a many-sorted signature, as well as Boolean connectives and quantifiers. The semantics is defined in terms of multialgebras, which interpret patterns as subsets of their carriers. This leads to a ternary satisfaction relation between patterns, multialgebras and elements (or states) of multialgebras.

Unlike first-order logic, matching logic is difficult to formalise faithfully as an institution due to the ternary nature of its satisfaction relation and to the fact that patterns are classified by sorts, much in the way the sentences of branching temporal logics are classified into state or path sentences and evaluated accordingly. We overcome this limitations by relying on the concept of stratified institution developed in [6], which extends institutions with an abstract notion of model state and defines a parameterised satisfaction relation that takes into account the states of models. We further develop this concept by adding classes,

which are determined by signatures, associated with sentences, and parameterise both the stratification of models and the satisfaction relation. We show that both matching and computation-tree logic can be described as stratified institutions with classes, and we adapt the canonical construction of an ordinary institution from a stratified one presented in [6] to take into consideration the role of classes.

The main advantage of using stratified institutions with classes to formalise matching logic is that we can extend the construction of reachability logic described in [4] from matching to other logical systems. Reachability logic is a formalism for program verification through which transition systems that correspond to the operational semantics of programming languages can be described using reachability rules; these rules rely on patterns and generalise Hoare triples in order to specify transitions between program configurations (similarly to term-rewrite rules). Therefore, reachability logic can be seen as a language-independent alternative to the axiomatic semantics and proof systems particular to each language. In our work, we define an abstract institution of reachability logic over an arbitrary stratified institution with classes such that by instantiating this parameter with matching logic we recover the original notion of reachability.

Having both matching and reachability logic defined as institutions allows us to integrate them into the logic graphs of institution-based heterogeneous specification languages such as HetCasl [7]. As an immediate result, the $\mathbb{K}$ framework can inherit the powerful module systems developed for specifications built over arbitrary institutions, with dedicated operators for aggregating, renaming, extending, hiding and parameterising modules. In addition, this will enable us to combine reachability logic and the tool support provided by $\mathbb{K}$ with other logical systems and tools. Towards that end, as a preliminary effort to integrate the $\mathbb{K}$ framework into Hets [8], we describe comorphims from matching and reachability logic to the institution of first-order logic.

References

1. Roşu, G., Şerbănuţă, T.F.: K overview and SIMPLE case study. Electronic Notes in Theoretical Computer Science **304**(0) (2014) 3–56
2. Şerbănuţă, T.F., Arusoaie, A., Lazar, D., Ellison, C., Lucanu, D., Roşu, G.: The K primer (version 3.3). Electronic Notes in Theoretical Computer Science **304**(0) (2014) 57–80
3. Roşu, G.: Matching logic: A logic for structural reasoning. Technical Report http://hdl.handle.net/2142/47004, University of Illinois (Jan 2014)
4. Roşu, G., Ştefănescu, A., Ciobâcă, Ş., Moore, B.M.: One-path reachability logic. In: Proceedings of the 28th Symposium on Logic in Computer Science (LICS'13), IEEE (June 2013) 358–367
5. Goguen, J.A., Burstall, R.M.: Institutions: Abstract model theory for specification and programming. J. ACM **39**(1) (1992) 95–146
6. Aiguier, M., Diaconescu, R.: Stratified institutions and elementary homomorphisms. Information Processing Letters **103**(1) (2007) 5–13
7. Mossakowski, T.: HetCasl–heterogeneous specification. Language summary (2004)
8. Mossakowski, T., Maeder, C., Lüttich, K.: The heterogeneous tool set, Hets. In Grumberg, O., Huth, M., eds.: TACAS. Volume 4424 of Lecture Notes in Computer Science., Springer (2007) 519–522

A Theoretical Foundation for Programming Language Aggregation ⋆

Ştefan Ciobâcă[1], Dorel Lucanu[1], Vlad Rusu[2], and Grigore Roşu[1,3]

[1] "Alexandru Ioan Cuza" University, Romania
[2] Inria Lille, France
[3] University of Illinois at Urbana-Champaign, USA

A **matching logic semantics** of a programming language is given by a tuple $\mathcal{L} = (\mathit{Cfg}, S, \Sigma, \Pi, \mathcal{T}, \mathcal{A}, \rightarrow_{\mathcal{T}})$, where $\Phi = (S, \Sigma, \Pi)$ is a first-order signature including the abstract syntax of the language as well as the syntax of the various operations in the needed mathematical domains, $\mathit{Cfg} \in S$ is a distinguished sort of **configurations**, $\mathcal{A}$ is a (possibly infinite) set of operational semantics rules of the language, $\mathcal{T}$ is the model of configurations of the language merged together with the needed mathematical domains, and the relation $\rightarrow_{\mathcal{T}}$ is the transition relation defined by the operational semantics. The semantic rules $\mathcal{A}$, also called **reachability rules**, are pairs $\varphi \Rightarrow \varphi'$ of matching logic formulae. For instance, the semantics of the assignment operator can be given by a rule of the form

$$\langle \texttt{x = } I\ C \rangle\, \langle S \rangle \wedge \mathit{isDefined}(\texttt{x}, S) \Rightarrow \langle C \rangle\, \langle \mathit{update}(S, \texttt{x}, I) \rangle$$

where C is the rest of the code to be executed and S is the store represented as a map from variables names to their values. The left-hand side and right-hand side of $\Rightarrow$ are examples of matching logic formulae.

As discussed in [2], conventional operational semantics of programming languages can be faithfully encoded as matching logic semantics.

The aggregation operation takes as input two language definitions $\mathcal{L}_L$ and $\mathcal{L}_R$ an returns a new definition $\mathcal{L}'$ capable of executing pairs of programs in the two languages according to their initial semantics. This operation is useful, e.g., in defining and proving equivalence of programs written in different languages or to prove the correctness of a compiler. In this paper we investigate the theoretical foundations for defining the aggregation operation for languages whose semantics is given by using matching logic.

The construction of $\mathcal{L}'$ is summarized by the following figure:

$$\begin{array}{ccccc}
(S_0, \Sigma_0, \Pi_0, \mathcal{T}_0) & \xrightarrow{h_R} & (\mathit{Cfg}_R, S_R, \Sigma_R, \Pi_R, \mathcal{T}_R, \mathcal{A}_R) & & \\
\Big\downarrow{\scriptstyle h_L} & & \Big\downarrow{\scriptstyle h'_R} & & \\
(\mathit{Cfg}_L, S_L, \Sigma_L, \Pi_L, \mathcal{T}_L, \mathcal{A}_L) & \xrightarrow{h'_L} & (S, \Sigma, \Pi, \emptyset, \mathcal{T}) & \xrightarrow{\iota} & (\mathit{Cfg}', S', \Sigma', \Pi', \mathcal{T}', \mathcal{A}')
\end{array}$$

⋆ This paper is supported by the Sectorial Operational Programme Human Resource Development (SOP HRD), financed from the European Social Fund and by the Romanian Government under the contract number POSDRU/159/1.5/S/137750.

It is known that the category of the first-order signatures admits push-outs, so (S, Σ, Π) is given by the foliowing push-out diagram:

$$\begin{array}{ccc} (S_0, \Sigma_0, \Pi_0) & \xrightarrow{h_R} & (S_R, \Sigma_R, \Pi_R) \\ {\scriptstyle h_L}\downarrow & & \downarrow{\scriptstyle h'_R} \\ (S_L, \Sigma_L, \Pi_L) & \xrightarrow{h'_L} & (S, \Sigma, \Pi) \end{array}$$

The model $\mathcal{T}' = \mathcal{T}$ is obtained by the amalgamation theorem, for which we give a constructive proof. The configurations in Cfg' are pairs of configurations $Cfg_L \times Cfg_R$ and the axioms $\mathcal{A}'$ includes the rules $\mathcal{A}_L \uplus \mathcal{A}_R$, modified such that they are applied now on the new configurations.

We also establish an extensionality result: a reachability rule $\varphi \Rightarrow \varphi'$ is a semantic consequence of $\mathcal{A}'$ iff its projections (reachability rules in the original languages) are semantic consequences of $\mathcal{A}_L$ and $\mathcal{A}_R$, respectively.

The aggregated language is useful, for example, to prove equivalence of programs. If we have a pair of programs $\langle S_L, S_R \rangle$ that both compute the sum of the first n natural numbers:

$S_L \equiv$ `s := 0; for i := 1 to n do s := s + i`
$S_R \equiv$ `let sum i = if i == 0 then 0 else i + sum (i - 1) in sum n`

We can show that S_L and S_R are equivalent (compute the same result) by showing a partial correctness property of the aggregated program $\langle S_L, S_R \rangle$: $\langle S_L, S_R \rangle \Rightarrow \langle s \mapsto n(n+1)/2, n(n+1)/2 \rangle$. Showing partial correctness can be done using reachability logic [3] or Hoare-like logics over the aggregated language. Based on the aggregated language, we have also developed a method of proving mutual equivalence [1].

A challenge is to organise the matching logic-based semantics of programming languages as a category and to investigate if the aggregated language can be obtained as a colimit in this category.

References

1. Ş. Ciobâcă, D. Lucanu, V. Rusu, and G. Roşu. A Language-Independent Proof System for Mutual Program Equiv. (rev.). Technical Report 14-01, UAIC, 2014.
2. G. Roşu and A. Ştefănescu. Checking reachability using matching logic. In *OOPSLA*, pages 555–574. ACM, 2012.
3. G. Roşu, A. Ştefănescu, Ştefan Ciobâcă, and B. M. Moore. One-path reachability logic. In *LICS*, pages 358–367, 2013.

Coq as a dependently typed system to build safe concurrent programs

June 30, 2014

Concurrent programs are notoriously hard to write correctly. They are difficult to test and to reason about, mainly because of their non-deterministic nature. Many type systems have been proposed in an attempt to write safe-by-construction concurrent programs. Safety properties include termination (in particular the absence of dead-lock) and the absence of execution error. Following the type based approach to safety, some type systems have been added to theoretical languages such as the π-calculus [3]. Others have been embedded into existing, strongly typed, general purpose languages such as *Haskell* [2] or *Idris* [1], through typed concurrent primitives. We present a novel approach to type parallel programs, encoding concurrent primitives into an existing dependently typed and purely functional language.

We chose the language *Coq*, based on the *Calculus of Inductive Constructions*. This calculus exploits the *Curry-Howard* isomorphism to combine proofs and programs in an elegant manner. *Coq* is a mature system, recipient of the ACM Software System Award, and had been the subject of extensive research and applications for more than twenty years.

In most programming languages, if an expression e has a type A it implies that "if the expression terminates and has no error, then its value is of type T". But e may never return a correct value, and we have no information about the side-effects made by e. On the contrary, *Coq* is a purely functional language. If an expression e has a type A then it terminates without error and returns a value of type A with no side-effect. Still, we can encode effects mimicking the way imperative effects are handled in *Haskell* with monads. In *Coq*, dependent types simplify the description of precise effects. We will define an effects system in *Coq* with concurrency, in the aim to lead to safer programming constructs.

To represent a concurrent *computation* with effects returning a value of type A, we define a new type $\mathcal{C}_{S,E}\,A$ in *Coq*:

$$\mathcal{C}_{S,E}\,A = S \to S \times (A + E + \mathcal{C}_{S,E}\,A)$$

A *computation* takes as argument a state of type S and returns a new state plus one of these three results:

- a value of type A if the computation terminated without error;
- an error of type E;
- a new computation of type $\mathcal{C}_{S,E}\,A$ (a pause point) representing the remaining work to be done to terminate the evaluation. This gives us intermediate steps to pause the evaluation of a computation.

A first key aspect is that the type $\mathcal{C}_{S,E}\,A$ is inductively defined so, if we iterate the evaluation steps, a computation will always terminate to a value or an error. A second key aspect is that the concurrency is represented using the pause mechanism. A computation with pauses explicits the steps leading to the final result, describing how the state evolves during evaluation. In particular, we will define the concurrent execution of two computations sharing a state by interleaving their evaluations at the pause points.

We combine two kinds of computations in a commutative and associative way with the $\uplus$ operator:

$$\mathcal{C}_{S_1,E_1}\,A \uplus \mathcal{C}_{S_2,E_2}\,A = \mathcal{C}_{S_1\times S_2,E_1+E_2}\,A$$

Polymorphic effects are expressed using the $\forall$ operator of *Coq* over the state or the error types. We define an operator `bind` to sequence two computations:

$$\texttt{bind} : \forall\, S\,E\,A\,B,\ \mathcal{C}_{S,E}\,A \to (A \to \mathcal{C}_{S,E}\,B) \to \mathcal{C}_{S,E}\,B$$

We encode inputs / outputs and non-termination effects into the computations type. For concurrency, we define two primitive operators:

$$\begin{aligned} \texttt{par} &: \forall\, S\,E\,A\,B,\ \mathcal{C}_{\mathcal{H}\times S,E}\,A \to \mathcal{C}_{\mathcal{H}\times S,E}\,B \to \mathcal{C}_{\mathcal{H}\times S,E}\,(A\times B)\\ \texttt{atomic} &: \forall\, S\,E\,A,\ \mathcal{C}_{S,E}\,A \to \mathcal{C}_{S,E}\,A \end{aligned}$$

The `par` operator executes two computations in parallel and returns the pair of outputs. The scheduling is preemptive and the thread switches are driven by an entropy state $\mathcal{H}$, an infinite stream of booleans. The `atomic` operator enforces the atomicity of a computation, hiding the intermediate steps of evaluation.

The complete implementation is available on `https://github.com/clarus/concurrent-computations`. We implemented other typed operators, like the parallel iteration of a function f with effects over a list l:

$$\texttt{iter\_par} : \forall\, S\,E\,A,\ (A \to \mathcal{C}_{\mathcal{H}\times S,E}\,\texttt{unit}) \to \texttt{list}\,A \to \mathcal{C}_{\mathcal{H}\times S,E}\,\texttt{unit}$$

We also implemented a *TODO-list* manager, communicating with a virtual user-interface and a server through an event system. All events are handled concurrently, and threads share a mutable data model using atomic transactions.

As a future work, we aim to execute efficiently expressions in $\mathcal{C}_{S,E}\,A$ by compilation to an existing concurrent programming language. For now, we simulate executions with the `eval` function:

$$\texttt{eval} : \forall\, S\,E\,A,\ \mathcal{C}_{S,E}\,A \to S \to S \times (A + E)$$

The definability of the `eval` function in *Coq* formally proves that all programs in $\mathcal{C}_{S,E}\,A$ terminate. The `eval` function also proves that errors can only be of type E. In particular programs in $\mathcal{C}_{S,\emptyset}\,A$ are error-free.

References

[1] Edwin Brady and Kevin Hammond. Correct-by-construction concurrency: Using dependent types to verify implementations of effectful resource usage protocols. 2010.

[2] Simon P. Jones. *Beautiful Concurrency.* O'Reilly Media, Inc., 2007.

[3] Naoki Kobayashi. Type systems for concurrent programs. 2003.

Heterogeneous refinement in HETS

Mihai Codescu and Till Mossakowski

Otto-von-Guericke University of Magdeburg

Refinement is an important concept in specification theory [6], relating abstract requirement specifications with more specific design specifications and implementations.

In [8] we introduced an extension of CASL [5] with a simple refinement language that adds means to formalize entire developments of software systems in the form of refinement trees, which capture the structure of implementations of the system. The refinement language generalizes and subsumes the architectural specification language of CASL. The difference is that architectural specifications specify only branching points in the development tree, while the refinement language allows for more flexibility in combining trees, including further refinement of some of their leaves. The language has been complemented in [2, 3] with calculi for checking correctness and consistency of refinements and is supported by the Heterogeneous Tool Set HETS [10].

The language definition follows the CASL convention of orthogonality between different layers of the language. That is, refinements are defined over an arbitrary institution that should satisfy very mild conditions. The structuring of specifications is also independent of the refinement language. The most basic form of refinement is refinement between specifications. Refinements can then be combined to form chains of refinements, or we can record the decision to decompose the task of implementing a specification into smaller subtasks by writing a refinement to an architectural specification. We can flexibly record further decisions regarding the design of a component either directly in the architectural specification (by allowing the specification of a unit to be itself a refinement) or after the architectural specification was written (by further refining its components).

In practice, it is natural that the formalisms used at different levels of the refinement trees of a software system should be different. Indeed, a logic allowing for more expressivity may be better suited for fixing the requirements, while a formalism closer to a programming language will appear closer to the leaves of the tree, to facilitate generation of code that should be correct by construction. One example detailed in [7] is the generation of C or Java code from a system of UML diagrams (where each type of UML diagram is formalised as a different institution).

Foundations for heterogeneous specification over a graph of institutions and translations include Grothendieck institutions [4], that allows to combine specifications written in different logics to end up in one logical systems, and heterogenous logical environments [9], that allow specifications written in different formalisms to coexist and be interlinked in a so-called distributed specification.

In this paper, we generalize the refinement language of CASL to the heterogeneous case, for the first time. We discuss which of the settings mentioned above, Grothendieck institutions and heterogeneous logical environments, is better suited for this purpose. Making some features of the language heterogeneous is a very straightforward task in both situations - we can easily capture refinement with change of logic as a refinement along a heterogeneous signature morphism. Other constructs raise more difficulties. To make architectural specifications heterogeneous, one has to solve not only the problem of discharging statically amalgamability conditions in their semantics [5] but also to deal with the fact that colimits of heterogeneous diagrams can only be approximated in the general case [1].

References

1. M. Codescu and T. Mossakowski. Heterogeneous colimits. In F. Boulanger, C. Gaston, and P.-Y. Schobbens, editors, *MoVaH'08 Workshop on Modeling, Validation and Heterogeneity*. IEEE press, 2008.
2. M. Codescu and T. Mossakowski. Refinement trees: calculi, tools and applications. In A. Corradini and B. Klin, editors, *Algebra and Coalgebra in Computer Science, CALCO'11*, volume 6859 of *Lecture Notes in Computer Science*, pages 145–160. Springer, 2011.
3. Mihai Codescu, Till Mossakowski, Donald Sannella, and Andrzej Tarlecki. Specification refinements: calculi, tools and applications. Submitted.
4. R. Diaconescu. Grothendieck institutions. *Applied categorical structures*, 10:383–402, 2002.
5. Peter D. Mosses (Ed.). CASL *Reference Manual*. LNCS 2960 (IFIP Series). Springer, 2004.
6. Hartmut Ehrig and Hans-Jörg Kreowski. Refinement and implementation. In Egidio Astesiano, Hans-Jörg Kreowski, and Bernd Krieg-Brückner, editors, *Algebraic Foundations of Systems Specification*, IFIP State-of-the-Art Reports, pages 201–242. Springer, 1999.
7. Alexander Knapp, Till Mossakowski, and Markus Roggenbach. An institutional framework for heterogeneous formal development in uml, 2014. CoRR abs/1403.7747.
8. T. Mossakowski, D. Sannella, and A. Tarlecki. A simple refinement language for CASL. In Jose Luiz Fiadeiro, editor, *WADT 2004*, volume 3423 of *LNCS*, pages 162–185. Springer; Berlin, 2005.
9. T. Mossakowski and A. Tarlecki. Heterogeneous logical environments for distributed specifications. In A. Corradini and U. Montanari, editors, *WADT 2008*, volume 5486 of *Lecture Notes in Computer Science*, pages 266 –289. Springer, 2009.
10. Till Mossakowski, Christian Maeder, and Klaus Lüttich. The Heterogeneous Tool Set. In Orna Grumberg and Michael Huth, editors, *TACAS 2007*, volume 4424 of *Lecture Notes in Computer Science*, pages 519–522. Springer-Verlag Heidelberg, 2007.

Term Graph Rewriting using Spans*

Andrea Corradini[1], Fabio Gadducci[1], and Tobias Heindel[2]

[1] [andrea,fabio](at)di.unipi.it, Dipartimento di Informatica, Università di Pisa, Italia
[2] theindel(at)inf.ed.ac.uk, School of Computer Science, University of Edinburgh, UK

So far, only a few categorical approaches to the rewriting of term graphs have been proposed, all of them based either on the Single- (SPO) or on the Double-Pushout Approach (DPO) to graph transformation. The goal was essentially to rephrase in a more declarative and algebraic way the classical set-theoretical, algorithmic approach by Barendregt et al. [1], relating term rewriting with the transformation of directed (typically acyclic) graphs, where the sharing of sub-terms, commonly used in implementations of functional languages, can be made explicit. The idea was also to apply in this framework general results already developed for the algebraic approaches to graph transformation, e.g. those concerning parallelism and concurrency.

The most relevant contributions in this line include [7], which is based on the DPO approach in the category $\mathbb{HG}$ of hypergraphs; and [8, 2], which are based on the SPO and the DPO approach, respectively, in the category $\mathbb{TG}$ of term graphs, considered as a sub-category of $\mathbb{HG}$. All such approaches share with [1] the result of soundness with respect to term rewriting (any term graph reduction using some encoding of term rules can be simulated by term rewriting). As [1], they also lack completeness, i.e. there could be term reductions that are not possible on term graphs, due to the presence of sharing. In fact, if two identical sub-terms happen to be shared in a term graph, both of them have to be rewritten in exactly the same way; instead, two identical sub-terms of a term could be rewritten using different rules.

To fix this weakness, one would need term graph rules able to "unfold" or "unravel" a term graph, creating copies of shared sub-terms. This paper presents some preliminary results addressing this problem, exploiting the Sesqui-Pushout (SqPO) approach, originally introduced in [4]. In this approach, a *rule* $p = (L \leftarrow l - K - r \rightarrow R)$ consists of a span of morphisms l and r, which are not necessarily monic. Given a *match*, i.e. a morphism $L - m \rightarrow G$, a *transformation* $G \Rightarrow_{p,m} H$ from G to H exists if the diagram in Fig. 1 can be constructed, such that (1) is a *final pullback complement* (FPBC), and (2) is instead a pushout.

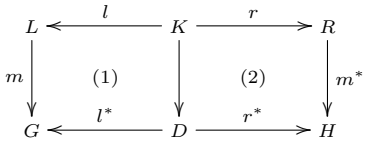


Fig. 1: A SqPO derivation.

The main feature of SqPO is the ability to specify the cloning or copying of structures, which is impossible both in the SPO (by definition) and in the DPO (as it would make rewriting non-deterministic). Furthermore, for left-linear rules it subsumes both the DPO and the SPO, under mild conditions.

* The work of the first and of the second author has been partially supported by the EU FP7-ICT IP ASCENS and by the MIUR PRIN CINA.

Unlike the *pushout complement* characterization of the left square (1) in the diagram in Fig. 1, which is used in the classical DPO approach, its characterization as *final pullback complement* is defined by a universal property, thus if it exists it is necessarily unique. General conditions for the existence of pushouts in the category of term graphs are discussed for example in [2]. In this note we focus instead on conditions for the existence of final pullback complements.

We first show that SqPO rewriting is possible for arbitrary term graph rules if matches are *regular monos*, also by exploiting the fact that $\mathbb{TG}$ is *rm-adhesive* [6] (also known as *quasi-adhesive* [9]), as shown in [3]. This has little applicability, though, as regular monos must preserve empty nodes, which informally means that variables in a rule cannot be instantiated. Nevertheless, even if for arbitrary monic matches SqPO rewriting is not possible in general, we are able to show that it works if the nodes duplicated by the rule are empty, i.e. have no outgoing edges. This is sufficient for some unsharing rules we are interested in.

Finally, we report on more complex conditions that are sufficient for the existence of FPBCs, as well as on conditions guaranteeing that a FPBC actually is a pushout complement as well, a condition required in the recently introduced *reversible sesqui-pushout approach* [5].

References

1. Barendregt, H., van Eekelen, M., Glauert, J., Kennaway, J., Plasmeijer, M., Sleep, M.: Term graph reduction. In: de Bakker, J., Nijman, A., Treleaven, P. (eds.) PARLE 1987. LNCS, vol. 259, pp. 141–158. Springer (1987)
2. Corradini, A., Rossi, F.: Hyperedge replacement jungle rewriting for term rewriting systems and logic programming. Theoretical Computer Science 109, 7–48 (1993)
3. Corradini, A., Gadducci, F.: On term graphs as an adhesive category. In: Fernandez, M. (ed.) TermGraph 2004. vol. 127(5), pp. 43–56 (2005)
4. Corradini, A., Heindel, T., Hermann, F., König, B.: Sesqui-pushout rewriting. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 30–45. Springer (2006)
5. Danos, V., Heindel, T., Honorato-Zimmer, R., Stucki, S.: Reversible sesqui-pushout rewriting for NACs. In: König, B., Giese, H. (eds.) ICGT 2014. LNCS, vol. 8571, pp. 161–176. Springer (2014)
6. Garner, R., Lack, S.: On the axioms for adhesive and quasiadhesive categories. Theory and Applications of Categories 27(3), 27–46 (2012)
7. Hoffmann, B., Plump, D.: Implementing term rewriting by jungle evaluation. RAIRO - Theoretical Informatics and Applications 25, 445–472 (1991)
8. Kennaway, R.: Graph rewriting in some categories of partial morphisms. In: Ehrig, H., Kreowski, H.J., Rozenberg, G. (eds.) Graph-Grammars and Their Application to Computer Science 1990. LNCS, vol. 532, pp. 490–504. Springer (1991)
9. Lack, S., Sobociński, P.: Adhesive and quasiadhesive categories. RAIRO - Theoretical Informatics and Applications 39(3), 511–545 (2005)

Two institutions of finite-state methods

Tim Fernando

Trinity College Dublin, Ireland
Tim.Fernando@tcd.ie

Abstract. Goguen and Burstall's notion of an institution is applied to two declarative representations of finite-state methods, one based on Monadic Second-Order Logic, the other on Kripke semantics over deterministic accessibility relations. Two institutions are defined, with a view to relating and possibly combining them in useful ways.

1 Introduction

Introduced to cope with the proliferation of logical systems in computer science, the concept of an *institution* (Goguen and Burstall 1992) has been studied intensely and applied widely (e.g. Diaconescu 2012). At the center of an institution are relations $\models_\Sigma$ of *satisfaction* between *models* and *sentences* of various *signatures* Σ, organized in a category, relative to which models and sentences are presented through functors. This set-up is applied below to two declarative representations of finite-state methods, one based on Monadic Second-Order Logic (MSO), the other on Kripke semantics for modal logic. Two institutions, $\mathcal{I}_1$ and $\mathcal{I}_2$, are outlined in turn, with a view to relating and possibly combining them in useful ways. (Separately, $\mathcal{I}_1$ is useful for conceptions of time given by runs of automata, and $\mathcal{I}_2$ for object-oriented knowledge bases).

2 $\mathcal{I}_1$: reducts and compression

For any finite alphabet Σ, Büchi's Theorem (e.g. Theorem 7.21, page 124, Libkin 2010) says a language $L \subseteq \Sigma^+$ is regular iff for some sentence φ in the set MSO_Σ of MSO-sentences over Σ,

$$L \;=\; \{s \in \Sigma^+ \mid s \models_\Sigma \varphi\}.$$

Now, for any subset A of Σ, the set MSO_A of MSO-sentences over A is a subset of MSO_Σ, and we might expect to reduce instances of $\models_\Sigma$ given by MSO_A to $\models_A$, mapping a string $s \in \Sigma^+$ to a string $f_A(s) \in A^+$ such that

$$\text{for any } \varphi \in \mathrm{MSO}_A, \qquad s \models_\Sigma \varphi \iff f_A(s) \models_A \varphi.$$

However, no such function f_A exists; the $\mathrm{MSO}_{\{a\}}$-sentence $\forall x\ P_a(x)$ (putting a at every string position x) is satisfied by all strings in $\{a\}^+$ but not all in $\{a,b\}^+$. The problem is, however, easy enough to fix; expand the alphabet Σ of

strings against which sentences in MSO_Σ are interpreted to the set 2^Σ of subsets of Σ (i.e., the power set of Σ), allowing into a string position any number of symbols $a \in \Sigma$. Redefining satisfaction $\models^\Sigma$ between $(2^\Sigma)^+$ and MSO_Σ, we have whenever $A \subseteq \Sigma$ and $s \in (2^\Sigma)^+$,

(†) for any $\varphi \in \mathrm{MSO}_A$, $\quad s \models^\Sigma \varphi \iff \rho_A(s) \models^A \varphi$

where $\rho_A(s)$ is the A-reduct of s, intersecting s componentwise with A

$$\rho_A(\alpha_1 \cdots \alpha_n) \;:=\; (\alpha_1 \cap A) \cdots (\alpha_n \cap A)$$

(Fernando 2014). For any set Φ large enough so that Σ belongs to the set $\mathit{Fin}(\Phi)$ of finite subsets of Φ, (†) is the satisfaction condition for the institution with

(i) signature category $\mathit{Fin}(\Phi)$ partially ordered by $\subseteq$
(ii) sentence functor $\mathit{sen}_0 : \mathit{Fin}(\Phi) \to \mathbf{Set}$ mapping $A \in \mathit{Fin}(\Phi)$ to MSO_A, and (A, B) such that $A \subseteq B \in \mathit{Fin}(\Phi)$ to the inclusion $\mathit{sen}_0(A, B) : \mathrm{MSO}_A \hookrightarrow \mathrm{MSO}_B$ $(\varphi \mapsto \varphi)$, and
(iii) model functor $\mathit{Mod}_0 : \mathit{Fin}(\Phi)^{op} \to \mathbf{Set}$ mapping $A \in \mathit{Fin}(\Phi)$ to $(2^A)^+$, and (B, A) such that $A \subseteq B \in \mathit{Fin}(\Phi)$ to $\mathit{Mod}_0(B, A) : (2^B)^+ \to (2^A)^+$, $s \mapsto \rho_A(s)$.

For infinite models (including the real line $\mathbb{R}$, with Φ chosen so that $\mathbb{R} \subseteq \Phi$), it is useful to compose ρ_A with *block compression* bc, which given a string s, compresses blocks α^n of $n > 1$ consecutive occurrences in s of the same symbol α to a single α, leaving s otherwise unchanged

$$bc(s) \;:=\; \begin{cases} bc(\alpha s') & \text{if } s = \alpha\alpha s' \\ \alpha\, bc(\beta s') & \text{if } s = \alpha\beta s' \text{ with } \alpha \neq \beta \\ s & \text{otherwise.} \end{cases}$$

Observe that $bc(s)$ is stutter-free, where a string $\alpha_1\alpha_2 \cdots \alpha_n$ is *stutter-free* if $\alpha_i \neq \alpha_{i+1}$ for i from 1 to $n-1$. If each symbol α_i is construed as a stretch of time, bc implements the slogan "no time without change" (with $bc(s) = s$ iff s is stutter-free). Now, for the institution $\mathcal{I}_1$, we keep the same signature category, but adjust the remaining components, defining $\mathit{sen}_1(A) = 2^A \times \mathrm{MSO}_A$, $\mathit{sen}_1(A, B) : \mathit{sen}_1(A) \hookrightarrow \mathit{sen}_1(B)$, $\mathit{Mod}_1(A) = \{bc(s) \mid s \in (2^A)^+\}$, $\mathit{Mod}_1(B, A) : \mathit{Mod}_1(B) \to \mathit{Mod}_1(A)$, $s \mapsto bc(\rho_A(s))$, and

$$s \models_1^\Sigma (C, \varphi) \quad\iff\quad bc(\rho_C(s)) \models^\Sigma \varphi$$

for $s \in \mathit{Mod}_1(\Sigma)$ and $(C, \varphi) \in \mathit{sen}_1(\Sigma)$, respecting the satisfaction condition.

3 $\mathcal{I}_2$: labelled deterministic transitions

Fix a large set Lab of labels. The objects of the signature category $\mathbf{Sign}_2$ of $\mathcal{I}_2$ are regular languages $L \subseteq \mathit{Lab}^+$, and the $\mathbf{Sign}_2$-morphisms from signatures L to L' form the hom-set

$$\mathbf{Sign}_2(L, L') \;=\; \{(L, l, L') \mid l \in \mathit{Lab}^* \text{ and } (\forall l' \in L)\; ll' \in L'\}$$

using the null string for identities, and concatenating for composition. For $(L, l, L') \in \mathbf{Sign}_2(L, L')$, the label l is applied to

(i) a modal operator $\langle l \rangle$ by $\mathcal{I}_2$'s sentence functor $sen_2 : \mathbf{Sign}_2 \to \mathbf{Set}$

$$sen_2(L, l, L') : sen_2(L) \to sen_2(L'),\ \varphi \mapsto \langle l \rangle \varphi$$

and to

(ii) deterministic transitions in a partial function $F : Q \times Lab^* \rightharpoonup Q$ (induced in reverse from its restriction to $Q \times Lab$, specifying Lab-atomic steps) by $\mathcal{I}_2$'s model functor $Mod_2 : {\mathbf{Sign}_2}^{op} \to \mathbf{Set}$

$$\begin{aligned} Mod_2(L) &= \{q \in Q \mid (\forall l \in L)\ (q, l) \in domain(F)\} \\ Mod_2(L', l, L) &: Mod_2(L') \to Mod_2(L),\ q' \mapsto F(q', l) \end{aligned}$$

making the satisfaction condition

$$q' \models_2^{L'} \langle l \rangle \varphi \quad \Longleftrightarrow \quad F(q', l) \models_2^{L} \varphi$$

for $q' \in Mod_2(L')$ and $\varphi \in sen_2(L)$, as in the Kripke semantics for $\langle l \rangle$.

References

R. Diaconescu 2012. Three decades of institution theory. In Jean-Yves Beziau (ed.), *Universal logic: an anthology*, pages 309 – 322, Springer.

T. Fernando 2014. Incremental semantic scales by strings, TTNLS, ACL archive, http://anthology.aclweb.org//W/W14/W14-1408.pdf.

J.A. Goguen and R.M. Burstall 1992. Institutions: Abstract Model Theory for Specification and Programming, *J. ACM* 39: 95–146.

L. Libkin 2010. *Elements of Finite Model Theory*. Springer.

(Co)algebraic semantics of heavy-weighted automata

Marie Fortin[1,2], Marcello Bonsangue[2,3], and Jan Rutten[2,4]

[1] *École Normale Supérieure de Cachan, France*
[2] *Centrum Wiskunde & Informatica (CWI), The Netherlands*
[3] *LIACS - Leiden University, The Netherlands*
[4] *Radbound University Nijmegen, The Netherlands*

June 30, 2014

Weighted automata are a generalization of non-deterministic automata in which each transition carries a weight [4]. This weight is an element of a semiring, representing, for example, the cost or probability of taking the transition. Weighted automata have many different areas of application, such as language processing, speech recognition, and image processing. More recently, weighted automata have been used to solve counting problems, first in [5] with a procedure called *coinductive counting*, then in [3] with the *counting automata methodology*.

Whereas non-deterministic automata either accept or reject a word, weighted automata associate with each word the cost of its execution. Their semantics is thus defined in terms of *weighted language* (also called formal power series), which are functions mapping words to weights.

Weighted languages form themselves a semiring, and thus can be chosen as the set of weights. We obtain *heavy-weighted automata*: automata in which each transition is labeled by a weighted language.

Such automata have already been introduced in [3], where they are called *counting automata*, and are used to give a compact representation of some combinatorial problems. One of our motivations here is to adapt the definitions given of *counting automata* to a coalgebraic setting, already existing for traditional weighted automata (see e.g. [5], [1]), and to try to formalize a more general framework for the reduction of some well-shaped infinite automata, based on the examples given in [3].

Our second motivation to the introduction of heavy-weighted automata is to provide something similar to what generalized automata (where transitions are labeled by regular expressions) are to ordinary automata. In particular, we will see that Brzozowski-McCluskey's *state elimination method* ([2],[7]) to compute the regular expression associated with a finite automaton also works for weighted automata.

Though heavy-weighted automata can be seen as ordinary weighted automata over the semiring of weighted languages, their semantics as such (given in terms of weighted languages over the semiring of weighted languages) is not so interesting. Instead, we want to define the semantics of heavy-weighted automata in terms of (ordinary) weighted languages. We propose four equivalent ways to define it:

- by giving a system of equations linking the semantics of the different states
- by first computing the semantics in terms of weighted languages over the semiring of weighted languages, and then mapping it to a weighted language with ordinary weights
- in terms of final homomorphisms of coalgebras. The set of weighted languages is the final coalgebra for the functor $S \times (-)^A$, which means that from any $S \times (-)^A$-coalgebra, there is a unique homomorphism of coalgebras to the set of all weighted languages. Heavy-weighted automata are not themselves $S \times (-)^A$-coalgebras, but they can be embedded into one, using some kind of determinization procedure (as introduced in [6]). We can then use the final $S \times (-)^A$-homorphism to define their semantics.
- by giving a procedure that transform a heavy-weighted automaton into an ordinary weighted automaton. This is done by composing ordinary weighted automata that recognise the weighted languages labeling the transitions of the heavy-weighted automaton.

The second part of our contribution consists in some examples of the usefulness of heavy-weighted automata. First, we give an adaptation of Brzozowski and McCluskey's state elimination method to weighted automata, which allows us to simplify a weighted automaton and, in the finite case, to compute a regular expression for the weighted language it recognises. Secondly, we show how heavy-weighted automata can be used to give a finite representation of some well-shaped, infinite weighted automata recognising algebraic weighted languages.

References

[1] F. Bonchi, M. M. Bonsangue, M. Boreale, J. J. M. M. Rutten, and A. Silva. A coalgebraic perspective on linear weighted automata. *Inf. Comput.*, 211:77–105, 2012.

[2] J. Brzozowski and J. Mccluskey, E. J. Signal flow graph techniques for sequential circuit state diagrams. *Electronic Computers, IEEE Transactions on*, EC-12(2):67–76, April 1963.

[3] R. D. Castro, A. Ramírez, and J. L. Ramírez. Applications in enumerative combinatorics of infinite weighted automata and graphs. *Scientific Annals of Computer Science*, 24(1), 2014.

[4] M. Droste, W. Kuich, and H. Vogler. *Handbook of Weighted Automata*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[5] J. J. M. M. Rutten. Coinductive counting with weighted automata. *Journal of Automata, Languages and Combinatorics*, 8(2):319–352, 2003.

[6] A. Silva, F. Bonchi, M. M. Bonsangue, and J. J. M. M. Rutten. Generalizing determinization from automata to coalgebras. *Logical Methods in Computer Science*, 9(1), 2013.

[7] D. Wood. *Theory of Computation*. Harper & Row, 1987.

Herbrand's Theorem in Hybrid Institutions

Daniel Găină

Research Center for Software Verification
Japan Advanced Institute of Science and Technology (JAIST)
daniel@jaist.ac.jp

Abstract

Hybrid logics [1] are a brand of modal logics that allows direct reference to the possible worlds/states in a simple and very natural way through the so-called nominals. This feature has several advantages from the point of view of logic and formal specification. For example, it becomes considerably simpler to define proof systems in hybrid logics [2], and one can prove results of a generality that is not available in non-hybrid modal logic. In specifications of dynamic systems the possibility of explicit reference to specific states of the model is an essential feature. The hybrid logical framework considered here is very general, formalised internally to abstract institutions [5].

In this paper we investigate a series of model-theoretic properties of hybrid logics in an institution-independent setting such as basic set of sentences [3], substitution [4] and reachable model [9, 8]. While the definition of basic set of sentences is a straightforward extension from a basic institution to its hybrid counterpart, the notion of substitution needs much consideration. Establishing an appropriate concept of substitution is the most difficult part of the whole enterprise to construct an initial model of a given hybrid theory and prove a variant of Herbrand's theorem.

Initial semantics [6] is closely related to good computational properties of logical systems and it plays a crucial role for the semantics of abstract data types and of logic programming. For example, initiality supports the execution of specification languages through rewriting, thus integrating efficiently formal verification of software systems into modelling. Our approach to initiality is layered and is intimately linked to the structure of sentences [7]. The existence of initial models of sets of atomic sentences is assumed in abstract setting but is developed in concrete examples. Then we show that the initiality property is closed to certain constructors for sentences.

The second main contribution of the paper is a variant of Herbrand's theorem for hybrid institutions, which reduces the satisfiability of a query with respect to a hybrid theory to the search of a suitable substitution. The institution-independent status of the present study makes the results applicable to a multitude of (concrete) hybrid logics including those obtained from hybridisation of non-conventional logics used in computer science.

References

1. P. Blackburn. Representation, reasoning, and relational structures: a hybrid logic manifesto. *Logic Journal of the IGPL*, 8(3):339–365, 2000.
2. T. Braüner. Hybrid Logic and its Proof-Theory. In *Applied Logic Series*, volume 37. Springer, 2011.
3. R. Diaconescu. Institution-independent Ultraproducts. *Fundamenta Informaticæ*, 55(3-4):321–348, 2003.
4. R. Diaconescu. Herbrand Theorems in arbitrary institutions. *Inf. Process. Lett.*, 90:29–37, 2004.
5. J. Goguen and R. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, 1992.
6. J. A. Goguen and J. W. Thatcher. Initial algebra semantics. In *SWAT (FOCS)*, pages 63–77. IEEE Computer Society, 1974.
7. D. Găină and K. Futatsugi. Initial semantics in logics with constructors. *Journal of Logic and Computation*, 2012. doi: 10.1093/logcom/exs044.
8. D. Găină, K. Futatsugi, and K. Ogata. Constructor-based Logics. *Journal of Universal Computer Science*, 18(16):2204–2233, aug 2012.
9. D. Găină and M. Petria. Completeness by Forcing. *J. Log. Comput.*, 20(6):1165–1186, 2010.

On the Semantics of HELENA Ensemble Specifications «extended abstract»

Annabelle Klarl and Rolf Hennicker

Ludwig-Maximilians-Universität München
Germany**

Advanced software systems involve a large number of autonomic, possibly heterogeneous components collaborating for some global goal in a distributed and highly dynamic environment. The EU project ASCENS [1,3] develops foundations, techniques and tools to support the whole life cycle of such systems (Autonomic Service Component ENSembles). In this context, we have developed the HELENA approach [2], which provides a formal model for the specification of ensembles as goal-oriented communication groups on top of a component-based platform. A conceptual key point of HELENA is that components can play different *roles* and that they can participate (under certain roles) in several ensembles, possibly at the same time. By adopting specific roles, components can join and leave ensembles as needed and they can change their behavior according to the required functionalities in a particular collaboration.

HELENA is based on a rigorous typing discipline, distinguishing between types and instances. Component instances, classified by *component types*, are considered as carriers of basic information relevant across many ensembles. Whenever a component instance joins an ensemble, the component adopts a role by creating a new role instance and assigning it to itself. The kind of roles a component is allowed to adopt is determined by *role types* which specify a set of role specific attribute types and a set of message types supported by the role for collaboration. To define the structural characteristics of collaborations, we use *ensemble structures*. They define which role types (constrained by multiplicities) are needed in a collaboration and determine which role types may interact by which message types. The idea is that collaboration is performed by communication between role instances; component instances do not directly interact, but they can be accessed by their owning role instances. In HELENA, an *ensemble specification* $EnsSpec = (\Sigma, RoleBeh, init)$ consists of an *ensemble structure* Σ, a set $RoleBeh$ of *role behavior specifications*, and a predicate *init* describing the admissible initial states of an ensemble. We provide a role behavior specification $RoleBeh_{rt}$ for each role type rt occurring in Σ. It specifies the life cycle of each role instance of the type rt. Role behaviors are represented by a simple process algebra with constructs for the null process, action prefix, nondeterministic choice, and recursion. There are actions for creating and retrieving role instances, sending or receiving messages, and performing internal computations (for instance, accessing the owning component).

** This work has been partially sponsored by the EU project ASCENS, 257414.

A first idea towards a formal semantics of ensemble specifications has been formulated in [2] using *ensemble automata* as semantic objects. An ensemble automaton formalizes the evolution of an ensemble in terms of a labeled transition system. An ensemble state records (1) the currently existing role instances, (2) which component instance currently adopts which role instance(s), (3) the data currently stored by the existing instances, and (4) the current control state of each role instance showing its local progress of execution. The transitions of an ensemble automaton are labeled either by communication labels or by management labels. Communication labels express collaboration of role instances by message exchange, management labels express creation of new role instances or retrieval of existing ones when needed for a collaboration.

In this work we propose, for each ensemble structure Σ, a satisfaction relation between ensemble specifications and ensemble automata formed over Σ. The satisfaction relation relies (1) on the satisfaction of the *init* predicate by the initial state of the automaton and (2) on a family of relations, one for each eventually existing role instance ri of type rt, between the control states of ri and the states obtained from the operational semantics of the process expression given by the behavior specification $RoleBeh_{rt}$. From the relations we can derive that any action performed by an ensemble automaton must be allowed by the local steps of the behavior specifications given for each role type. Then we define the model class semantics of an ensemble specification $EnsSpec = (\Sigma, RoleBeh, init)$ leading to a loose semantics of ensemble specifications. In a next, step we show how a canonical model can be constructed from the ensemble specification using a set of rules which derive a maximal set of admissible execution steps of the ensemble from the local role behavior specifications. We show that this canonical model is initial in the class of all models of the specification. Initiality is based on an appropriate homomorphism notion between ensemble automata. It relies on a kind of simulation relation between the states of the two automata taking into account (1) an injective mapping between the sets of eventually existing instances of each automaton and (2) a family of relations between their control states. Our semantics relies on a synchronous communication scheme, but it should be straightforward to generalize it to asynchronous communication.

References

1. The ASCENS Project (2014), `http://www.ascens-ist.eu`
2. Hennicker, R., Klarl, A.: Foundations for Ensemble Modeling - The Helena Approach. In: Iida, S., Meseguer, J., Ogata, K. (eds.) Specification, Algebra, and Software. Lecture Notes in Computer Science, vol. 8373, pp. 359–381. Springer (2014)
3. Wirsing, M., Hölzl, M., Tribastone, M., Zambonelli, F.: ASCENS: Engineering Autonomic Service-Component Ensembles. In: Beckert, B., Damiani, F., Bonsangue, M., de Boer, F. (eds.) 10th International Symposium on Formal Methods for Components and Objects. Lecture Notes in Computer Science, vol. 7542. Springer (2012)

An Institutional Framework for Heterogeneous Formal Development in UML

Alexander Knapp[1], Till Mossakowski[2], and Markus Roggenbach[3]

[1] Universität Augsburg
[2] Otto-von-Guericke-Universität Magdeburg
[3] Swansea University

In industrial software design, the Unified Modeling Language (UML) is the predominately used development mechanism. In aerospace industry, e.g., the company AEC uses the UML to define the software architecture of aeroplane engine controllers through various levels of abstraction from a layered architecture overview to a detailed class, operation and attribute definition of the software components. This model is then used for code generation.

The UML is an OMG standard [8], which describes a language family of 14 types of diagrams, of structural and behavioural nature. A typical development by AEC easily involves eight different UML diagrams. The OMG specification provides an informal semantics of nine sub-languages in isolation. The languages are mostly linked through a common meta-model, i.e., through abstract syntax only. This situation leads to a gap between standards' recommendation to apply formal methods, and current industrial practice, which by using the UML lacks the semantic foundations to apply such methods. One common approach to deal with this gap is to define a comprehensive semantics for the UML using a system model, e.g., [1,2]. However, this is a thorny business, as every detail has to be encoded into one, necessarily quite complex semantics. Furthermore, such an approach has difficulties to cater for UML's variations of usage, leading to company or domain-specific variations. In our work, we outline a competing approach by providing a heterogeneous semantics based on institutions [5], where we extend [4] by considering a subset of diagrams rich enough for industrial use.

The languages and UML diagram types that we consider are shown in Fig. 1. On the *modelling level* we use parts of the UML and the Object Constraint Language (OCL). On the *implementation level* we currently employ the programming language C and ACSL. For substantial fragments of several UML diagram types, we have already provided a formalisation as institutions:

Class diagrams in [4], we have sketched an institution for class diagrams, which has been detailed in [6]. It includes a construction for stereotypes.

Component diagrams form an institution similar to that for class diagrams. The main difference are the connector types, which however are quite similar to associations.

Object diagrams are essentially reifications of models of class diagrams.

Composite structure diagrams are similar to object diagrams. The main difference are the connectors, which however are quite similar to the links of object diagrams.

Interactions in [4], we have sketched an institution for interactions, as well as their interconnection (also with class diagrams) via institution comorphisms.

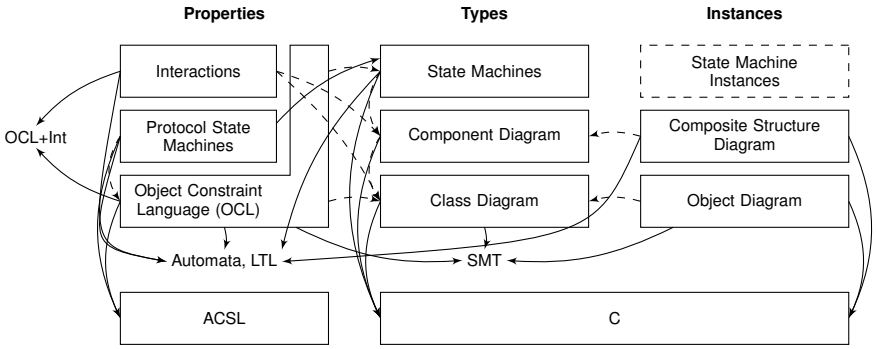


Fig. 1. Institution morphisms (dashed arrows) and institution co-morphisms (solid arrows) between the languages and diagrams

OCL in [4], we have sketched institutions for OCL. In [3], the OCL semantics is presented in more detail. An institution based on this is in preparation.

Thus, the central remaining challenge for institutionalisting UML are **state machines** and **protocol state machines**, **C** and **ACSL** as institutions.

We distinguish between diagrams for properties, types and instances, where we express the meaning of a model in a sub-language/diagram directly in an appropriate semantic domain, especially by providing an institution for state machines. We further systematically identify meaningful connections given by the abstract syntax of the UML specification or which can be gleaned from its semantic description. The separation between the meaning of the individual diagrams and their relation allows our approach to be adopted by different methodologies, for instance an object-oriented approach or a component-based one.

References

1. M. Broy, M. V. Cengarle, H. Grönniger, and B. Rumpe. Considerations and Rationale for a UML System Model. In Lano [7], chapter 3, page 43–60.
2. M. Broy, M. V. Cengarle, H. Grönniger, and B. Rumpe. Definition of the System Model. In Lano [7], chapter 4, page 61–93.
3. M. V. Cengarle and A. Knapp. OCL 1.4/5 vs. 2.0 Expressions — Formal Semantics and Expressiveness. *Softw. Syst. Model.*, 3(1):9–30, 2004.
4. M. V. Cengarle, A. Knapp, A. Tarlecki, and M. Wirsing. A Heterogeneous Approach to UML Semantics. In *Concurrency, Graphs and Models*, LNCS 5065. Springer, 2008.
5. J. A. Goguen and R. M. Burstall. Institutions: Abstract Model Theory for Specification and Programming. *J. ACM*, 39:95–146, 1992.
6. P. James, A. Knapp, T. Mossakowski, and M. Roggenbach. Designing Domain Specific Languages — A Craftsman's Approach for the Railway Domain Using CASL. In *WADT'12*, LNCS 7841. Springer, 2013.
7. K. Lano, editor. *UML 2 — Semantics and Applications*. Wiley, 2009.
8. Object Management Group. Unified Modeling Language. Standard, OMG, 2011.

What is a derived signature morphism?

Ulf Krumnack[1], Till Mossakowski[2], and Tom Maibaum[3]

[1] University of Osnabrück, Germany
[2] Otto-von-Guericke University of Magdeburg, Germany
[3] McMaster University, Hamliton, Canada

The notion of signature morphism is basic to the theory of institutions. It provides a powerful primitive for the study of specifications, their modularity and their relations in an abstract setting. The notion of *derived* signature morphism generalises signature morphisms to more complex constructions, where symbols may be mapped not only to symbols, but to arbitrary terms. Derived signature morphisms have been introduced in [6] and studied in [9, 7, 10, 1, 2]. Recently, the notion of derived signature morphism has gained attention in the field of model-driven engineering [3], where it is closely related to the notion of query. Queries also play a role in logic programming [8], databases and ontological engineering [4]. Furthermore, derived signature morphisms may enhance approaches that currently only consider plain signture morphisms or some form of substitution, e.g. in analogy making [11] and conceptual blending [5].

The purpose of this work is to study derived signature morphisms in an institution-independent way. The motivation is to give an institution independent semantics to the notion of derived signature morphism, query and substitution in the context of the Distributed Ontology, Modeling and Specification Language DOL, which currently being standardised as an OMG standard, as which will serve a meta-language for expressing logical theories and their relations in the application areas mentioned above.

We will recall two known approaches to derived signature morphisms, introduce a third one, and discuss their pros and cons:

1. The first approach is to consider derived signature morphisms to be ordinary signature morphisms into a conservative or definitional extension [].
2. The second approach is to consider derived signature morphisms to be abstract morphisms that induce abstract substitutions in the sense of [1, 2]. The crucial property is to introduce sentence translations and model reducts.
3. The third approach is to consider Kleisli institutions, which have derived signature morphisms as signature morphisms. A Kleisli institution is defined for an institutional monad, which is a monad in the 2-category of institutions, institutions morphisms and discrete institution morphism modifications. The Kleisli institution then is the Kleisli object (in the sense of 2-category theory) of the institutional monad.

The first approach is very general and works in any (weakly) semi-exact institution. While models can be reduced against derived signature morphisms, the drawback is that sentences cannot be translated along them. dblp johnson The second approach remedies this problem axiomatically: model reducts and sentence translation are required to exist. Moreover, powerful Herbrand theorems relate queries and substitutions [1, 2].

The third approach is more specific about the nature of derived signature morphisms: they are obtained through a Kleisli construction in a monad, which provides a more precise (abstract) description of what derived signature morphisms are. In particular, it allows to study important question, e.g. which colimits exist in these various categories of derived signature morphisms.

On a more general level, this approach shows again, that notions from basic category theory (monads and kleisli construction) can be adopted to institutions and lead to useful concepts there. It naturally leads to the question, if related notions, like the Eilenberg-Moore construction, can give raise to meaningful applications in an institutional setting as well.

References

1. R. Diaconescu. Herbrand theorems in arbitrary institutions. *Information Processing Letters*, 90:29–37, 2004.
2. R. Diaconescu. *Institution-Independent Model Theory*. Birkhäuser, 2008.
3. Zinovy Diskin, Tom Maibaum, and Krzysztof Czarnecki. Intermodeling, queries, and Kleisli categories. In Juan de Lara and Andrea Zisman, editors, *FASE'12 Proceedings of the 15th international conference on Fundamental Approaches to Software Engineering*, volume 7212 of *Lecture Notes in Computer Science*, pages 163–177. Springer, 2012.
4. Birte Glimm, Carsten Lutz, Ian Horrocks, and Ulrike Sattler. Conjunctive query answering for the description logic shiq. *J. Artif. Intell. Res. (JAIR)*, 31:157–204, 2008.
5. Joseph Goguen and D. Fox Harrell. Style: A computational and conceptual blending-based approach. In Shlomo Argamon, Kevin Burns, and Shlomo Dubnov, editors, *The Structure of Style: Algorithmic Approaches to Understanding Manner and Meaning*, pages 291–316. Springer, Berlin, 2010.
6. Joseph A. Goguen, James W. Thatcher, and Eric G. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In Raymond T. Yeh, editor, *Current Trends in Programming Methodology – Vol. IV: Data Structuring*, pages 80–149. Prentice-Hall, 1978.
7. Furio Honsell, John Longley, Donald Sannella, , and Andrzej Tarlecki. Constructive data refinement in typed lambda calculus. In *Proceedings of the 3rd International Conference on Foundations of Software Science and Computation Structures. European Joint Conferences on Theory and Practice of Software (ETAPS 2000)*, volume 1784 of *Lecture Notes in Computer Science*, pages 161–176, Berlin, 2000. Springer.
8. J.W. Lloyd. *Foundations of Logic Programming*. Springer Verlag, 1987.
9. Donald Sannella and Rod M. Burstall. Structured theories in lcf. In Giorgio Ausiello and Marco Protasi, editors, *Proceedings of the 8th Colloquium on Trees in Algebra and Programming (CAAP '83)*, pages 377–391. Springer, 1983.
10. Donald Sannella and Andrzej Tarlecki. *Foundations of Algebraic Specification and Formal Software Development*. Monographs in Theoretical Computer Science. Springer, Berlin, 2012.
11. Martin Schmidt, Ulf Krumnack, Helmar Gust, and Kai-Uwe Kühnberger. Heuristic-driven theory projection: An overview. In Henri Prade and Gilles Richard, editors, *Computational Approaches to Analogical Reasoning: Current Trends*, volume 548 of *Studies in Computational Intelligence*, pages 163–194. Springer, 2014.

Safety and Performance of the European Rail Traffic Management System: A Modelling and Verification Exercise in Real Time Maude

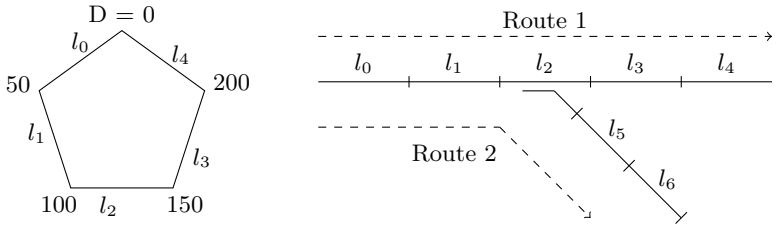
Andrew Lawrence, Ulrich Berger,
Markus Roggenbach, and Monika Seisenberger

Swansea University, UK
ajlawrence@acm.org
{u.berger,m.roggenbach,m.seisenberger}@swansea.ac.uk

ERTMS. The European Rail Traffic Management System is a next generation train control system loosely specified in [3]. Traditionally the railway has been a Boolean world of discrete signals, *track circuits* for train detection and *interlockings* which use propositional equations to guarantee train safety. In contrast, the ERTMS deals with continuous data that allows for a finer grain of control over the railway. Here, a *radio block centre* (RBC) grants each train a block of track called a *movement authority* in which the train is allowed to move. ERTMS requires trains to have on-board equipment that ensures trains to break in time. While the correctness of traditional interlocking systems is relatively well understood [1], it is ongoing research of how to verify ERTMS for safety properties such as collision freedom due to the involvement of continuous data. Further, it is an open field of how to substantiate the claim that the ERTMS approach offers a higher performance of the railway compared to traditional railway control.

Methodology. Real Time Maude is an extension of Full Maude which supports the specification to real-time and hybrid systems. Offered techniques include simulation and time-bounded LTL model checking for verification of timed properties. We use the Real Time Maude system [2] for the specification of ERTMS and the Maude linear temporal logic (LTL) model checker [4] for verification. In our model, speed, acceleration, braking behaviour and track length are integral parts. This allows us to specify and prove the safety of the system and to study the system performance via execution of the specification as a simulation.

Case Studies. We first formalise a simple example railway in the shape of a pentagon (see Fig. 1 (a)) with two trains, one slower than the other, an interlocking, an RBC and five track segments $\{l_0, \ldots, l_4\}$ as four hybrid automata to capture the behaviour of the system. This example demonstrates how the ERTMS system deals with a length of track containing multiple trains which are controlled by an RBC and interlocking. Then we proceed to model this small example in the Real Time Maude system as an object orientated specification capturing the message passing and communications between the different components of the system. This specification is executed to simulate the behaviour of the modelled railway. The faster train can be seen catching up with the slow train and then braking and waiting for authorisation before moving off again.



(a) Pentagon Example (b) Junction Example

Fig. 1: Case Studies

We verify that the movement authorities of the two trains do not overlap which is expressed by safety properties of the form "it is globally true that if both trains are behind their movement authorities then either train A's movement authority is behind train B's or visa versa".

As a second case study we model an open system, a simple junction (see Fig. 1 (b)) which contains a single point and two routes each consisting of five track segments. Trains are inserted onto the railway line according to a schedule by a controller object with a given route which they must follow. This example demonstrates the behaviour of ERTMS with respect to two further important constructs in the railway domain namely routes and points. It enables us to analyse the throughput of the ERTMS system through a junction where one train must wait for the point to become available. The Maude LTL model checker is then applied to verify that a point does not move in a given movement authority, a safety property that is essential for the prevention of derailment.

Results & Future Work. Though the presented case studies are of simple nature, they demonstrate that our modelling approach works: safety properties can be formulated and verified in reasonable time via model-checking, performance can be studied via simulation. It is future work to study realistic examples, possibly to develop abstractions, and also to establish performance properties as theorems.

References

1. Chadwick, J. , James, P. Kanso, K., Lawrence, A. Moller, F., Roggenbach, M., Seisenberger, M. and Setzer, A.: Verification of solid state interlocking programs. In SEFM'13, LNCS 8368, Springer 2014.
2. Ölveczky, P. C. and Meseguer, J.: Specification and Analysis of Real-Time Systems Using Real-Time Maude. In FASE'04. LNCS 2984, Springer 2004.
3. International Union of Railways: ETCS System Requirements Specification (SRS) ver. 2.3.0 (2006)
4. Eker, S., Meseguer, J., and Sridharanarayanan A.: The Maude LTL Model Checker and Its Implementation. In SPIN'03, LNCS 2648, Springer 2003.

A canonical proof-theoretic approach to model theory

Carlos G. Lopez Pombo[1,2], Paula D. Chocrón[1,2], Ignacio Vissani[1,2], and Tomas S.E. Maibaum[3]

[1] Department of Computing, FCEyN, Universidad de Buenos Aires
[2] Consejo Nacional de Investigaciones Científicas y Tecnológicas (CONICET)
[3] Department of Computing and Software, McMaster University

Logic has proved essential as a formal language for describing different aspects of software artefacts. These formal descriptions, frequently called specifications, have served not only as requirements documentation but also for proving properties, provided the logical language in which the specification is written has an appropriate reasoning tool. Semantics is an integral part of logic, as providing logical descriptions of real-world phenomena requires people to agree on how these descriptions should be interpreted. In this sense, model theory has been seen as providing the cornerstone for the satisfaction of this need. Model theory is usually understood as the study of classes of mathematical structures satisfying formulae in a formal language of choice. Model theory is a tool for characterising semantic notions, like meaning and truth, associated to syntactic objects, like formulae and proofs, of a corresponding language. From a category theory point of view the model theory of a logic has been formalised as an *institution* [1]. An institution is a structure $\langle \mathsf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \{\models_\Sigma\}_{\Sigma \in |\mathsf{Sign}|} \rangle$ formed by: 1) a category of signatures Sign, 2) a grammar functor $\mathbf{Sen} : \mathsf{Sign} \to \mathsf{Set}$ providing a set of sentences for each signature, 3) a model functor $\mathbf{Mod} : \mathsf{Sign}^{op} \to \mathsf{Cat}$ providing a class of semantic structures for each signature, 4) a family of binary relations $\{\models_\Sigma\}_{\Sigma \in |\mathsf{Sign}|}$ such that given $\Sigma \in |\mathsf{Sign}|$, $\models_\Sigma \subseteq |\mathbf{Mod}(\Sigma)| \times |\mathbf{Sen}(\Sigma)|$, and 5) satisfying that for all $\sigma : \Sigma \to \Sigma' \in ||\mathsf{Sign}||$, $\alpha \in \mathbf{Sen}(\Sigma)$ and $\mathcal{M} \in \mathbf{Mod}(\Sigma')$, $\mathcal{M} \models_{\Sigma'} \mathbf{Sen}(\sigma)(\alpha)$ if and only if $\mathbf{Mod}(\sigma)(\mathcal{M}) \models_\Sigma \alpha$.

In mathematical logic, given a logical system, we are forced to consider all possible semantic structures that can interpret its sentences, while in computer science we are mostly concerned about the analysability of the semantics as we rely on it to prove properties of software artefacts (as well as meta properties of the logic itself). Usually, we place trust only in those structures that can be described resorting to a formal logical language; typically they are maximal consistent theories in the language of choice, like those used in Henkin's completeness proof for equational logic [2], or theories over some formalisation of set theory. Our aim is to undertake a recasting of the notion of semantics in syntactic terms to support this approach.

Behavioural specifications, such as those written in any dynamic logic [3], temporal logics, both linear time [4], and branching time [5,6], etc., usually involve the following common elements: 1) an interpretation of a subset of symbols whose interpretation is fixed for all states usually referred to as rigid, 2) an ordering of states (for example, sequences of states in linear temporal logics, trees

of states in branching time temporal logics, a single state in dynamic logics, etc.) such that each of the constituent states provide an interpretation of a different subset of symbols, referred to as flexible symbols, and 3) a satisfaction relation providing meaning for behavioural logic operators. Generally, the ordering of states is obtained from a binary relation between them; for example, in dynamic logics there is a set of atomic actions and regular programs defined over them; in temporal logics, both linear time and branching time, there is a single transition relation; in deontic logics there are events produced by actions, etc.

This paper addresses the question of whether such a class of structures can be constructed in a canonical way so the definition of the functor **Mod**, in the definition of institutions, can be given in concrete representable terms.

Equational logic extended with extra-logical predicate symbols has been widely accepted as an appropriate specification language for describing the operations of abstract data types [7]. Equational theories can also be used to provide interpretations, of extra-logical symbols by considering formulae of the form $f(t_1, \ldots, t_n) = t$ and $P(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ and t are ground terms of the logical language of choice. On the other hand, we extend the *Elementary Theory of Binary Relations* [8] by incorporating the additional relational operators of ω-closure fork algebras [9]. This class of relation algebras have been used to reason about relations due to their complete (almost) equational calculus and its easy-to-understand concrete semantics, build out of a set of binary relations.

In this work we propose a general framework that facilitates the definition of the semantics of a logical system by identifying and properly characterising its static and dynamic properties. To make the approach flexible, we propose the use of a higher-order extension of equational logic to formalise the static aspects, while the dynamic properties characterising the accessibility relations between states are expressed by means of concrete models for fork algebraic terms.

References

1. Goguen, J.A., Burstall, R.M.: Institutions: abstract model theory for specification and programming. Journal of the ACM **39**(1) (1992) 95–146
2. Henkin, L.A.: The logic of equality. The American Mathematical Monthly **84**(8) (1977) 597–612
3. Harel, D., Kozen, D., Tiuryn, J.: Dynamic logic. Foundations of Computing. MIT Press.
4. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems. Springer-Verlag.
5. Emerson, E.A., Halpern, J.Y.: Decision procedures and expressiveness in the temporal logic of branching time. Jour. of Comp. and Syst. Sciences **30**(1) (1985) 1–24
6. Pnueli, A.: The temporal logic of programs. In: Proceedings of 18th. Annual IEEE Symposium on Foundations of Computer Science, Los Alamitos, CA, USA, IEEE Computer Society, IEEE Computer Society (1977) 46–57
7. Ehrig, H., Mahr, B., Orejas, F.: Introduction to algebraic specification: Formal methods for software development. Computer Journal **35**(5) (1992) 468–477
8. Tarski, A.: On the calculus of relations. Jour. of Symb. Logic **6**(3) (1941) 73–89
9. Frias, M.F.: Fork algebras in algebra, logic and computer science. Volume 2 of Advances in logic. World Scientific Publishing Co., Singapore (2002)

Quasiary Specification Algebras and Logics

Mykola Nikitchenko

Taras Shevchenko National University of Kyiv, Ukraine
nikitchenko@unicyb.kiev.ua

Algebraic approach to system specification has the following two characteristics: 1) the formalism of many-sorted algebras is used to model software systems; 2) special logics based on such many-sorted algebras are used to reason about system properties. In the literature various kinds of such algebras and logics are described (e.g. see [1]).

In this paper we present a special kind of algebras and logics defined for classes of quasiary mappings. Informally speaking, such mappings are partial mappings defined over partial states (partial assignments) of variables. Conventional n-ary mappings can be considered as a special case of quasiary mappings.

We identify the following properties of quasiary mappings: *partiality*, *sensitivity* to unassigned variables, and *unrestricted* (possibly infinite) *arity*.

These features, on the one hand, better reflect properties of software systems, but on the other hand, they complicate construction and investigation of logics of quasiary mappings comparing with classical logic oriented to n-ary mappings. In this case some laws of classical logic are violated. In particular, partiality of predicates violates *Modus Ponens, sensitivity* of predicates violates the law $\forall xP \to P$. Thus, construction and investigation of algebras and logics of quasiary mapping is an important challenge.

We start with the definition of quasiary many-sorted specification algebras. The arrows $\xrightarrow{p}$ and $\xrightarrow{t}$ define respectively classes of partial and total mapping. Let T be a *class of types* and $\tau : V \xrightarrow{t} \mathrm{T}$ be a total mapping called *type valuation*. Given V, T and τ, a class $NST(V, \mathrm{T}, \tau)$ (shortly: $NST(\tau)$) of *typed nominative sets* is defined by the formula $NST(\tau) = \tilde{\prod}_{v \in V} \tau(v)$, where $\tilde{\prod}_{v \in V} \tau(v)$ is the partial Cartesian product. Informally speaking, typed nominative sets represent partial states of typed variables. The class of *many-sorted partial quasiary predicates* is denoted $Pr(\tau) = NST(\tau) \xrightarrow{p} Bool$. Let $A \in \mathrm{T}$. The class of *many-sorted partial quasiary ordinary functions into* A is denoted $Fn^A(\tau) = NST(\tau) \xrightarrow{p} A$. The class of *many-sorted partial bi-quasiary functions* (*program functions*) is denoted $FPrg(\tau) = NST(\tau) \xrightarrow{p} NST(\tau)$.

We specify the following *quasiary specification program algebra*:

$$QSPA(\tau) =< Pr(\tau), \{Fn^A(\tau) \mid A \in \mathrm{T}\}, FPrg(\tau);$$
$$\vee, \neg, {}'x, S_P^{\bar{x}}, S_A^{\bar{x}}, \exists x, =_A, AS^x, id, \bullet, IF, WH, FH, \cdot >$$

with operations (compositions [2]) of disjunction $\vee$, negation $\neg$, parametric denomination $'x$, parametric superposition $S_P^{\bar{x}}$ of ordinary functions into a predicate ($\bar{x} = (x_1, ..., x_n) \in V^*$), parametric superposition $S_A^{\bar{x}}$ of ordinary functions into an ordinary function with the range A, parametric existential quantifier

$\exists x$, parametric equality $=_A$, parametric assignment AS^x, identity id, sequential execution $\bullet$; conditional IF, loop WH, and prediction $\cdot$ (sequential execution of a program function and a predicate [3]). In the case of deterministic functions the prediction composition is related to the weakest precondition defined by Dijkstra and possibility/necessity operations of dynamic logic. This composition provides possibility to construct predicates describing program properties. Using this composition we can define continuous Floyd-Hoare composition for partial predicates [4] by the formula $FH(p, pr, q) = p \to pr \cdot q$.

Restricting $QSPA(\tau)$ on the two first carriers (thus deleting program functions) we obtain a *quasiary predicate algebra* $QA(\tau)$ which can be considered generalization of first-order classical logic on quasiary predicates [5].

The class of *quasiary specification algebras* (for different τ) forms a semantic basis for quasiary specification program logics. We introduce the notion of logic signature Σ which includes the sets of variables V, predicate symbols Ps, ordinary function symbols Fs, and program functions symbols FPs (with corresponding type valuation mappings) and construct the logic language as the set of terms over Σ. The formula interpretation mapping is defined in a usual way.

The following results have been obtained:

(1) many-sorted quasiary specification program and predicate algebras and logics have been constructed and investigated;
(2) sound and complete sequent calculi have been constructed for many-sorted quasiary predicate logics;
(3) algorithms for reduction of the satisfiability problem in predicate logics and special subclasses of program logics to the satisfiability problem in classical logic have been developed;
(4) a quasiary program logic which can be considered as an extension of Floyd-Hoare logic on partial predicates has been defined and investigated; sound and extensionally complete calculi have been constructed for such logics.

It is expected that the obtained results will be able to serve as a basis for extending automated reasoning systems with more expressive logical languages.

References

1. Sannella, D., Tarlecki, A.: Foundations of Algebraic Specification and Formal Software Development. Springer (2012)
2. Nikitchenko, M., Tymofieiev, V.: Satisfiability in composition-nominative logics. Central European Journal of Computer Science, vol. 2, issue 3, 194–213 (2012)
3. Glushkov, V.: Automata theory and formal transformations of microprograms. Cybernetics, 5, 3–10 (1965) In Russian
4. Kryvolap, A., Nikitchenko, M., Schreiner, W.: Extending Floyd-Hoare logic for partial pre- and postconditions. CCIS, 412, 355–378, Springer (2013)
5. Nikitchenko, M., Shkilnyak, S.: Mathematical logic and theory of algorithms. Publ. house of Taras Shevchenko National Univ. of Kyiv, Kyiv (2008) In Ukrainian

Proving liveness properties using abstract state machines and n-visibility

Norbert Preining, Kokichi Futatsugi, and Kazuhiro Ogata

Japan Advanced Institute of Science and Technology
Research Center for Software Verification
Nomi, Ishikawa, Japan
{preining,futatsugi,ogata}@jaist.ac.jp

Introduction One methodology for formal specification of a complex system is using Abstract State Machine (Asm), where the system is described as a set of states with transitions between states. Although a very simple paradigm, it renders itself applicable in a wide variety of scenarios, from program analysis over protocol specification to design planning.

In the setting of formal, more specifically algebraic specification and verification, we aim at a set of conditions on the states, which we want to prove to hold in all reachable states.

In the following we describe the general methodology in proving properties using Asm. In particular, we extend one of the most common proof methods, namely induction over the set of reachable states, to include additional properties on transitions. This new proof method, we call it n-visibility (for $n \in \mathbb{N} \cup \{\infty\}$), allows for the verification of liveness properties, which by itself are out of the reach of the base proof method by induction on the set of reachable states.

We continue in defining an extended system that includes transitions, and show that the base proof method in the extended system and the proof method with n-visibility in the base system coincides. Using this method we can show that extension with infinite visibility acts as fix-point of this extension procedure, and can prove all properties that one can prove with ∞-visibility, including the typical liveness properties, but also more exotic fairness properties.

Abstract state machine Asm is a well known concept, we only recall a few definitions. Let us assume an algebra $\mathcal{A}$ of a many-sorted (order-sorted) signature Σ, and indicate with State the sort as well as the set of all ground terms representing states. Transitions are pairs of states (ground state terms), the set of all transitions is denoted by $\mathcal{T}$. With *system* we denote the set of algebra, state sort, initial states and transitions: $\Pi = (\mathcal{A}, \mathsf{State}, \mathcal{I}, \mathcal{T})$. The set of reachable states $\mathcal{R}(\Pi)$ is defined as usual via transition chains. Transition chains are defined as sequence of states that form a transition. We denote by $\mathcal{TC}^n(s)$ the set of transition chains of length n that start from a given state s.

The general aim of specification and verification is to prove that a property `inv` holds for the set of all reachable states. It is well known from the literature that in most cases the set of reachable states is not easily describable, or equivalently, it is not decidable whether a state is in $\mathcal{R}$ or not. If we still want to prove a property $\texttt{inv}(r)$ for all $r \in \mathcal{R}$, we have to use an inductive approach.

Theorem 1 (base proof method). *The property* $\mathtt{inv}$ *holds for all reachable states* r*, i.e.,* $\forall r \in \mathcal{R}\,\mathtt{inv}(r)$*, if the following two conditions are satisfied: (1)* $\mathtt{inv}(s)$ *holds for all* $s \in \mathcal{I}$*, and (2)* $\mathtt{inv}(s) \to \mathtt{inv}(s')$ *holds for all* $(s, s') \in \mathcal{T}$*.*

n-visibility We consider a proof method that allows not only look at a single state, but also at transitions between reachable states, in particular, n transitions starting from the current state into the future. Here we include the case of *infinite* visibility, i.e., that $n = \infty$. Assume that a predicate $\mathtt{prop}$ on transition sequences, in particular $\mathcal{TC}^n$, is given, the proof method with n-visibility for $n \in \mathbb{N} \cup \{\infty\}$ is defined as follows:

Theorem 2 (proof method with n-visibility). *Assume that* $\mathtt{inv}$ *is a property on states, and* $\mathtt{prop}$ *is a property on* $\mathcal{TC}^n$*. To show that* $\forall r \in \mathcal{R}\,\mathtt{inv}(r)$ *and* $\forall r \in \mathcal{R}\ \forall \tau \in \mathcal{TC}^n(r)\ \mathtt{prop}(\tau)$ *holds, it suffices to show the following properties: (1)* $\mathtt{inv}(s)$ *holds for all* $s \in \mathcal{I}$*, (2)* $\forall s \in \mathsf{State}\ \forall \tau \in \mathcal{TC}^n(s)\ \mathtt{inv}(s) \to \mathtt{prop}(\tau)$ *holds, and (3)* $\forall (s, s') \in \mathcal{T}\ \mathtt{inv}(s) \to \mathtt{inv}(s')$*.*

The *base proof method* only discusses properties of states, and is thus of restricted expressiveness. On the other hand, if we use the proof method with n-visibility we can include more general properties into the discussion, in particular *liveness* properties which often require a comparison of states.

Extension system The concept of visibility introduced above can be used to give an algebraic description and methodology, by including transition sequences into the description. Let $\Pi = (\mathcal{A}, \mathsf{State}, \mathcal{I}, \mathcal{T})$ be a state-based ASM. We defined the n-visibility extension of Π, in symbols $\xi^{(n)}(\Pi)$ by adding a new sort $\mathsf{State}^{(n)}$, a new constructor $u^{(n)}$ of type $u^{(n)} : \mathsf{State} \times \mathsf{State}^n \to \mathsf{State}^{(n)}$. Ground terms of the extended algebra are called *meta-states*.

It is possible to define transitions and initial states in the extended algebra such that they are compatible with the transitions and initial states in the base system with respect to transition sequences.

Based on this we can prove the following theorems:

Theorem 3. *Provability in* Π *with* n*-visibility coincides with (base) provability in* $\Pi^{(n)}$ *(i.e., without visibility).*

The above clearly shows that by considering extended systems we gain expressivity, as we are enabled to speak and prove properties about sets of transitions. We conclude with two results on iterated extension and its fixpoint:

Theorem 4. *Iterated extension with* n*-visibility adds up:*

$$\xi^{(n)}(\xi^{(m)}(\Pi)) = \xi^{(n+m)}(\Pi)$$

and the extension of Π *with* ∞*-visibility provides the strongest expressivity.*

We will also provide implementation of these concepts in CafeOBJ, proving liveness properties for QLOCK, and are planning to give a general mechanism for extending state space based specifications within CafeOBJ.

Improving the Quality of Use Cases via Model Construction and Analysis*

Leila Ribeiro, Érika Cota, Lucio Mauro Duarte, and Marcos A. de Oliveira

PPGC - Institute of Informatics – Federal University of Rio Grande do Sul (UFRGS)
PO Box 15.064 – 91.501-970 – Porto Alegre – RS – Brazil
{leila,erika,lmduarte,marcos.oliveira}@inf.ufrgs.br

Abstract. *Use Cases (UC)* are a popular model to specify system behaviour and is an important artifact for system design and evolution. UC quality impacts the overall system quality and defect rates. UCs are however detailed in natural language, leaving space to various issues related to imprecision, ambiguity and incompleteness. Keep the expressiveness of an informal description and choose a formalism that is flexible enough to represent the semantics defined by the stakeholders and allow systematic analysis is thus a challenge. We propose a methodology to enhance the quality of the UC by applying tool-supported analyses to a formal model derived from UC description. We adopt a Graph Transformation model (GT) and describe a method for modeling, analysis and improvement of the UC description using tools and techniques available for this formal specification model. Analysis can reveal ambiguities, inconsistencies, and missing requirements. The result is not only an improved UC but also the creation of a high-quality documentation of the system: a formal model of the UC (given by a graph transformation system). We propose a translation from UCs to GTs that requires only basic knowledge of graphs and UCs and is systematic, describing what the developer should do at each step. It is faster and cheaper than a formal inspection, avoiding several rounds of discussion, and more efficient than an informal inspection because the analyses are executed following a well-defined process, based on a formal model and aided by the AGG tool, which automatically produces the artifacts presented in the paper. Our goal is to detect problems and provide an initial diagnostic feedback. This is done via analysis of a series of Open Issues (OIs). We describe possible outcomes of each step, how to interpret OIs and their level of severity, considering real and potential errors. We believe that even more important than pointing out a problem is presenting some insight on what its cause could be and how harmful it could be if not fixed.

Keywords. Use Cases, Graph Transformation, Model Analysis.

A preliminary version of the paper can be found in
https://dl.dropboxusercontent.com/u/23087660/WADT2014-RibeiroCotaDuarteOliveira.pdf

* This work is part of the VeriTeS project, supported by FAPERGS and CNpq

A Refinement Procedure for Inferring Side-Effect Constructs

Adrián Riesco[1], Irina Măriuca Asăvoae[2], Mihail Asăvoae[2]

[1] Universidad Complutense de Madrid, Spain
ariesco@fdi.ucm.es
[2] VERIMAG/Universitè Joseph Fourier, France
{irina.asavoae, mihail.asavoae}@imag.fr

1 Introduction

In this paper we propose a refinement procedure of a previously introduced technique, in [5], which infers language constructs producing side-effects (i.e. instructions in some programming language which, upon execution, will induce memory updates). Our target application is to design generic program analysis tools (e.g. a program slicer) based on a meta-level analysis of the programming language semantic definition. This would allow a certain degree of parametrization of the program analysis, in general, and of the program slicing in particular. As such, subsequent modifications of the formal semantics of the language would not result in changes of the analyzer since the analyzer automatically incorporates the modifications. Our approach builds on the formal executable semantics of the language of interest, given as a rewriting logic theory [4, 2] and on the program to be analyzed.

Our program slicing methodology combines two steps: (1) a generic analysis of the formal executable semantics followed by (2) a data dependency analysis of the program. Step (1) is a fixpoint computation of the set of the smallest language constructs that have side-effects which, in turn step (2) is used to extract safe program slices based on a set of variables of interest. We instantiated this formal semantics-based methodology to intra- [5] and inter-procedural [1] program slicing. The intraprocedural slicing is based on the classical WHILE language augmented with a side-effect assignment and read/write statements, specified in Maude [2], while the interprocedural variant uses the WhileF language, an extension which includes scope declaration for variables and language statements for function call and return. However, both program slicing methods are based on a less generic assumption: the general memory update operation—the assignment statement—, has a fixed destination: its left-hand side. This is not necessarily generic as, for example, the family of the assembly languages use explicit memory operations (load/store) and arithmetic/logic operations (which update registers), with flexible destination placement in the language syntax. For example:

- in `gas` - the GNU assembler (and the default back-end of the standard `gcc` compiler), an instruction like `movl $0, %eax` copies the value `0` into the register `%eax`. The update is from left (source) to right (destination).

- in `x86` assembly language, an instruction like `mov ax, 0h` copies the value `0` into the register `ax`. The update is from right (source) to left (destination).

In this paper, we propose a refinement of our formal-semantics based program slicing which infers automatically the direction of the memory/storage update.

To make this inference, we make the standard assumption that equations are oriented from left to right, hence considering them as rewrite rules. Then the algorithm computes the set of basic language constructs which produce side-effects, by inspecting the conditions and the right-hand side of each rewrite rule in the definition. For this inspection, we employ unification and an adaptation of the backward chaining technique [6]. The results are used as contexts in step (2) to infer a safe program slicing as described in the followings. We start with the program, P and a set of variables of interest V. First we identify and label the contexts containing variables of interest and increment the set V with the other variables appearing in the currently identified context. We run this step until V stabilizes. At the end, the sliced program is represented by the skeleton term containing all the labeled contexts.

Our two steps slicing algorithm resembles the approach in [3], where an algorithm mechanically extracts slices from an intermediate representation of the language semantics definition. The algorithm relies on a well-defined transformation between a programming language semantics and this common representation. It also generalizes the notions of static and dynamic slices to that of constrained slices. What we propose is to eliminate the translation step to the intermediate representation and to work directly on the language semantics. Moreover, when we infer the direction of the memory update operation, we address, in an uniform way, a wide range of low-level languages.

References

1. Asavoae, I.M., Asavoae, M., Riesco, A.: Towards a formal semantics-based technique for interprocedural slicing. In: Integrated Formal Methods iFM (2014), to appear.
2. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L. (eds.): All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic, LNCS, vol. 4350. Springer (2007)
3. Field, J., Ramalingam, G., Tip, F.: Parametric program slicing. In: POPL. pp. 379–392 (1995)
4. Martí-Oliet, N., Meseguer, J.: Rewriting logic: roadmap and bibliography. Theor. Comput. Sci. 285(2), 121–154 (2002)
5. Riesco, A., Asavoae, I.M., Asavoae, M.: A generic program slicing technique based on language definitions. In: Martí-Oliet, N., Palomino, M. (eds.) WADT 2012. Lecture Notes in Computer Science, vol. 7841, pp. 248–264 (2013)
6. Sterling, L., Shapiro, E.Y.: The Art of Prolog - Advanced Programming Techniques. MIT Press (1986)

Solving Queries over Modular Logic Programs

Ionuţ Ţuţu[1,2] and José Luiz Fiadeiro[1]

[1] Department of Computer Science, Royal Holloway University of London
[2] Institute of Mathematics of the Romanian Academy,
Research group of the project ID-3-0439
ittutu@gmail.com, jose.fiadeiro@rhul.ac.uk

As in many other areas of computing science, in logic programming modularization or structuring techniques are an essential element in managing the inherent complexity of large software systems – in this case, logic programs – at various stages of their development. The literature on modularization in logic programming is vast, and it has evolved primarily along two somewhat divergent lines of research. Whereas a number of proposals have focused on augmenting logic-programming languages with module systems through dedicated constructs for building programs as hierarchical combinations of components, preserving in this way the denotational and the operational semantics of the underlying formalisms (see, e.g. [1,2,3]), others have explored the possibility of extending the base logic-programming language with new logical connectives so as to capture the operators needed for building and composing modules (see, e.g. [4,5]). A comprehensive survey of these general approaches is presented in [6].

Through our work we aim to bring the two aforementioned lines of research into harmony by studying modularization over a new model-theoretic framework for logic programming advanced in [7] that combines ideas specific to each of the directions. On the one hand, by extending Goguen and Burstall's concept of institution [8] with appropriate notions of variable, substitution, clause, query and goal-directed rule, the theory we propose accommodates logical systems whose expressive power goes beyond that of the original formalism of Horn-clause logic. On the other hand, inspired by recent developments in the theory of structured specifications [9], it defines logic programs in an axiomatic manner, as abstract structures characterized by signatures, sets of clauses and classes of models, capturing in this way not only representations of programs as plain sets of clauses [10], but also elaborate module systems like those described in [11,12].

Within this algebraic setting, we investigate two of the most fundamental aspects of the modularization of logic programs: the preservation and the reflection of solutions along morphisms of programs and, why not, along encodings of one logic-programming language, together with its associated module system, into another. From a computational point of view, these properties have a significant impact on the efficiency of searching for solutions to queries. In particular, the former gives us the possibility to search for solutions to queries in restricted contexts that correspond to subprograms or imported modules, and then translate these solutions back to the original setting, while the latter guarantees that all solutions can be obtained in this manner. It should be noted, however, that only the preservation property can be expected to hold for any morphism of logic programs, or any encoding of logic-programming languages. The reflection

property may hold for many morphisms or encodings of practical importance, but not for all; it is unlikely to hold, for instance, for those morphisms of logic programs that model simple implementations or refinements [13].

The contribution of the approach presented here is twofold. First, it provides a general framework for studying the modularization of logic programs independently of both their underlying logical systems and the chosen structuring mechanisms. In this sense, our work upgrades a number of institution-independent results reported in [14] by dropping or weakening requirements such as the liberality of signature morphisms and the existence of representable substitutions, as well as by axiomatizing the notion of logic program. Second, it distinguishes between correct and computed answers, i.e. between the denotational and the operational notions of solution to a query, enabling the computation of all solutions to certain queries even in situations in which the logic program under consideration is not query-complete, meaning that not every answer can be proved to be a solution using the program's goal-directed rules. This supports the development of more advanced notions such as parameterization and higher-order modules.

References

1. O'Keefe, R.A.: Towards an algebra for constructing logic programs. In: 1985 Symposium on Logic Programming, IEEE Computer Society (1985) 152–160
2. Goguen, J.A., Meseguer, J.: Eqlog: Equality, types, and generic modules for logic programming. In: Logic Programming: Functions, Relations, and Equations. Prentice Hall (1986) 295–363
3. Sannella, D., Wallen, L.A.: A calculus for the construction of modular Prolog programs. Journal of Logic Programming **12**(1&2) (1992) 147–177
4. Miller, D.: A logical analysis of modules in logic programming. Journal of Logic Programming **6**(1&2) (1989) 79–108
5. Giordano, L., Martelli, A., Rossi, G.: Extending Horn clause logic with implication goals. Theoretical Computer Science **95**(1) (1992) 43–74
6. Bugliesi, M., Lamma, E., Mello, P.: Modularity in logic programming. Journal of Logic Programming **19/20** (1994) 443–502
7. Ţuţu, I., Fiadeiro, J.L.: From conventional to institution-independent logic programming. (2013) submitted.
8. Goguen, J.A., Burstall, R.M.: Institutions: abstract model theory for specification and programming. Journal of the ACM **39**(1) (1992) 95–146
9. Diaconescu, R.: An axiomatic approach to structuring specifications. Theoretical Computer Science **433** (2012) 20–42
10. Lloyd, J.W.: Foundations of logic programming. Symbolic computation: Artificial intelligence. Springer (1987)
11. Sannella, D., Tarlecki, A.: Specifications in an arbitrary institution. Information and Computation **76**(2/3) (1988) 165–210
12. Borzyszkowski, T.: Logical systems for structured specifications. Theoretical Computer Science **286**(2) (2002) 197–245
13. Sannella, D., Tarlecki, A.: Toward formal development of programs from algebraic specifications: implementations revisited. Acta Informatica **25**(3) (1988) 233–281
14. Diaconescu, R.: Herbrand theorems in arbitrary institutions. Information Processing Letters **90**(1) (2004) 29–37

A Full Operational Semantics of Asynchronous Relational Networks*

Ignacio Vissani[1,2], Carlos G. Lopez Pombo[1,2],
Ionuţ Ţuţu[3,4], and José Luiz Fiadeiro[3]

[1] Department of Computing, FCEyN, Universidad de Buenos Aires, Argentina
[2] Consejo Nacional de Investigaciones Científicas y Tecnológicas, Argentina
[3] Department of Computer Science, Royal Holloway University of London, UK
[4] Institute of Mathematics of the Romanian Academy,
Research group of the project ID-3-0439
ivissani@dc.uba.ar, clpombo@dc.uba.ar,
ittutu@gmail.com, jose.fiadeiro@rhul.ac.uk

In the context of Service-Oriented Computing (SOC), the structure of software systems is intrinsically dynamic since the software applications that currently run over globally available computational and network infrastructures may require external services, and thus they may need to procure, on the fly, other applications that provide those services, and bind to them so that, collectively, given business goals can be fulfilled. In particular, development is no longer a process in which subsystems are developed and integrated by skilled engineers: in SOC, discovery and binding are performed, at run-time, by dedicated middleware.

In this talk, we provide an extension of the trace semantics of the service component algebra presented in [1] – Asynchronous Relational Networks (ARNs) – to account for the fact that, because of run-time discovery and binding, ARNs can be reconfigured at the same time as they compute. Our work follows the formalisation developed [2] in terms of hypergraphs whose nodes, or points, correspond to structured sets of messages that can be exchanged between the network components, and whose hyperedges capture those elements of networks that account for computation and/or communication, i.e. the processes and the channels attached to them through connections.

Every ARN determines sets of interaction-points whose associated messages can be regarded as information that is provided or required by the network. This supports a notion of execution of an ARN through which a network can discover and bind, at run time, to service modules of a given repository whenever it triggers an action related to the publication or the delivery of a requires-message.

Within this context, we define a new operational semantics for ARNs that relies on specialised forms of temporal-logic concepts such as execution fragment, path and trace. More precisely, we formalise the execution of an ARN as a sequence of networks and local states of their components; such an execution starts with the ARN under consideration and the initial states of its processes

* This work has been supported by the European Union Seventh Framework Programme under grant agreement no. 295261 (MEALS)

and channels, and unfolds through the effect of publication/delivery actions, which may trigger the discovery and binding of other networks.

This means that executions of an ARN can be regarded as sequences of generalised states connected by a transition relation, where a transition from one state to another can be the result of either the execution of an internal transition of the ARN, i.e. the effect of an action that is not associated with any of the ARN's provides- or requires-points, or the binding of one of the requires-points of the ARN to a provides-point of a service module taken from the repository as a consequence of executing an action associated with that requires-point.

The approach we propose here combines, in an integrated way, the operational semantics of processes and channels, and the dynamic reconfiguration of ARNs. It captures a full operational semantics of ARNs through labelled transition systems built from the local semantics of the considered networks, together with the semantics of those ARNs that are provided by a repository of services by means of stepwise execution, discovery and binding. This gives us a more refined view of the execution of ARNs than the logic-programming semantics of [2]. Moreover, it allows us to use various forms of temporal logic to express, and even to verify through standard model-checking techniques, properties concerning the behaviour of ARNs that are more complex than those considered before. It is possible, for example, to determine whether or not a given service module of a repository is used or may be used during the execution of an ARN, or to identify the differences between the nondeterministic behaviour of a component, reflected within the execution of an ARN, and the nondeterminism that arises from the discovery and binding to other ARNs.

References

1. Fiadeiro, J.L., Lopes, A.: An interface theory for service-oriented design. Theoretical Computer Science **503** (2013) 1–30
2. Ţuţu, I., Fiadeiro, J.L.: A logic-programming semantics of services. In Heckel, R., Milius, S., eds.: Algebra and Coalgebra in Computer Science. Volume 8089 of Lecture Notes in Computer Science., Springer (2013) 299–313

Fibred Amalgamation and Fibred Equivalences

Uwe Wolter / Harald König

University of Bergen, Norway / FHDW Hannover, Germany

Indexed semantics. Traditionally the semantics of formal specifications is defined in an "indexed" manner analogously to the concepts indexed set and indexed category, respectively. In *indexed semantics* we have the universe of "syntactic entities" on one side, an appropriate "semantic universe" on the other side and "semantic structures" are given by interpretations of syntactic entities in the semantic universe.

In the area of Algebraic Specifications the syntactic entities are traditionally algebraic signatures or specifications, respectively. The category **Set** of sets and total maps is the standard semantic universe and algebras are the corresponding semantic structures. First-order model theory and categorical algebra are other typical examples of "indexed semantics".

For the purpose of this talk we consider small (multi) graphs as our syntactic entities (signatures). In view of Algebraic Specifications these are nothing but many-sorted algebraic signatures with unary operation symbols only. As semantic universe we may chose the category **Set**. Note, that the category **Mult** would be more appropriate as semantic universe when we intend to define the semantics of the underlying graphs of class diagrams, for example (compare [5]). A "semantic structure" for a graph S is nothing but a graph homomorphism from the graph S into the underlying graph of the category **Set**.

Amalgamation in indexed semantics. Compositionality is a highly required property of any specification formalism [2]. Compositionality means essentially that we have amalgamation: The model functor, that assigns to a syntactic entity the category of all its interpretations in a fixed semantic universe, should map a pushout diagram in the category of syntactic entities to a pullback in the category of categories.

Amalgamation is kind of inherent for indexed semantics (namely, as long as our syntactic entities can be seen, in one or another way, as representations of categories): In our case of graphs as signatures, signature morphisms are just graph homomorphisms and the reduct functors are given by simple pre-composition of interpretations with signature morphisms. In such a way, amalgamation is trivially ensured for all (!) pushouts of signature morphisms by the fact that a pushout in the category **Graph** of small graphs is also a pushout in the category **GRAPH** of graphs.

Fibred semantics. In software engineering and, especially, in (meta) modelling we are faced, however, with "fibred semantics": Syntactic and semantic entities live in the same universe thus an entity may play, at the same time, a syntactic and

a semantic role (as in meta-modelling for example [5]). "Semantic structures" in fibred semantics are *instances* of syntactic entities that is morphisms in the common universe (*base category*) with a syntactic entity as target.

An instance of a graph S, for example, is given by a graph G and a graph homomorphisms $\iota : G \to S$ thus the category $Inst(S)$ of all instances of a graph S is just the slice category $\mathbf{Graph}/S$.

At a first glance it may be astonishing but by moving from indexed to fibred semantics we loose some properties that are essential for abstract model theory along the line of institutions [3].

The reduction of instances along a signature morphism is given by pullback construction. Thus, assuming an arbitrary but fixed choice of pullbacks in our base category, any signature morphism from $\varphi : S_1 \to S_2$ induces a forgetful functor from $Inst(S_2)$ to $Inst(S_1)$. Since the construction of pullbacks is only functorial up to isomorphism we get, however, on the global level not a model functor but only a model pseudo functor and this pseudo functor depends on an arbitrary but fixed choice of pullbacks (compare [1]).

Amalgamation in fibred semantics. In fibred semantics we don't have amalgamation for arbitrary pushouts but only for those pushouts that are *van-Kampen squares* [4]. In most base categories that appear in applications a sufficient condition for van-Kampen squares is that one of the legs of the span, constituting the pushout, is a monomorphism. This observation triggered the concept of *adhesive categories* [4].

Adhesive categories have been served as a conceptual basis for quite a lot of theoretical work on graph and model transformations. Restricting our investigations to adhesive categories, however, has some drawbacks. First, we ignore all the pushouts that are van-Kampen squares even if there are no monomorphisms involved. Second, there are practical relevant situations where exactly those "ignored van-Kampen squares" are needed.

To make all the van-Kampen squares, that are ignored by adhesive categories, available for practical applications we conducted a thorough analysis of van-Kampen squares in arbitrary topoi based on the concept of *descent data* [6]. For pre-sheaf topoi (and thus for **Set**, **Graph**, ...) we proved a necessary and sufficient condition for a pushout to be a van-Kampen square.

Recent work and talk. It appears that descent data can be equivalently described by fibred equivalences. Since equivalence and congruence relations are traditionally an essential basis for the theory of algebraic specifications this observation gives us a chance to build a bridge from our former general results to the area of algebraic specifications.

The talk is intended to cover the following topics:

- Definition and discussion of the concept of fibred equivalence.
- Reformulation and discussion of our former results in terms of fibred equivalences.

– Presentation of new results. Using fibred equivalences we have been able, for example, to show for arbitrary topoi that for any monic signature morphism the correspond (pullback based) reduct functor has a left adjoint.

References

1. Zinovy Diskin and Uwe Wolter. A Diagrammatic Logic for Object-Oriented Visual Modeling. *ENTCS*, 203/6:19–41, 2008.
2. H. Ehrig, M. Große–Rhode, and U. Wolter. Applications of category theory to the area of algebraic specification in computer science. *Applied Categorical Structures*, 6(1):1–35, 1998.
3. J. A. Goguen and R. M. Burstall. Institutions: Abstract Model Theory for Specification and Programming. *Journals of the ACM*, 39(1):95–146, January 1992.
4. S. Lack and P. Sobociński. Adhesive Categories. In I. Walukiewicz, editor, *proceedings of FOSSACS 2004*, pages 273–288. Springer, LNCS 2987, 2004.
5. Adrian Rutle, Alessandro Rossini, Yngve Lamo, and Uwe Wolter. A formal approach to the specification and transformation of constraints in MDE. *Journal of Logic and Algebraic Programming*, 81/4:422–457, 2012.
6. Uwe Wolter and Harald König. Fibred Amalgamation, Descent Data, and Van Kampen Squares in Topoi. *Applied Categorical Structures*, pages 1–40, 2013.

Fixed Point Logics as Institutions

Kiouvrekis Yiannis[1] and Stefaneas Petros[2]

[1] Department of Mathematics
School of Applied Mathematics and Applied Physical Sciences
National Technical University
Athens, 15780 GREECE
yiannisq@central.ntua.gr

[2] Department of Mathematics
School of Applied Mathematics and Applied Physical Sciences
National Technical University
Athens, 15780 GREECE
petros@math.ntua.gr

Abstract. It has been already proved that several logics are institutions [3],[14], such as **PL** - Propositional calculus, **FOL** - First Order Logic, $\boldsymbol{FOL^1}$ - Single-sorted logic, $\boldsymbol{FOL^+}$ - Positive First Order Logic, **UNIV** - Universal setences in first order logic, **HCL** - Horn clause logic, **EQL** - Equational logics, $(\Pi \cup \Sigma)_n^0$, **SOL** - Second order logic, $\boldsymbol{FOL_{\infty,\omega}}, \boldsymbol{FOL_{\alpha,\omega}}$ - infinitary logics , **MFOL** - Modal (first order) logic.

Finite model theory is the study of logics on classes of finite structures. One of the central issues in finite model theory is the relationship between logical definability and computational complexity. The expressive power of First Order Logic (FOL) is weak to express several properties on finite structures, like connectivity on finite graphs. Another critical issue is that first-order logic is not closed under inductive definition. For example, consider the notion of a connected component of a graph. We define this concept inductively. Let v a vertex of a graph G and $P_0(v) = \{v\}$, for each $n \in \mathbb{N}$ we define $P_n = \{x \in G \mid G \models R(x, y)$for some y $\in P_{n-1}(v)\}$. If G is finite graph then for some m $P_m(v) = P_{m+1}(v)$. In this case $P_m(v)$ is the connected component of v in G. Although first-order logic can define the sets P_m for each $m \in \mathbb{N}$, it cannot define the notion of a connected component. Hence first-order logic is not closed under inductive definitions. A methodology that can be followed in order to construct Logics with greater expressive power than FOL, should handle inductive definitions.

A way of modeling recursive definitions is to incorporate an explicit fixed point operator. Logics following this approach are called fixed-point logics. We consider logics that include various fixed-point operators. These logics are minimal extensions of first-order logic that are closed under inductive definitions.

There is more than one way to make the notion of inductive definition precise. Each corresponds to a different fixed-point operator.

1. Least Fixed Point Logic ($\boldsymbol{LFP}$)

2. Monotone Fixed Point Logic ($\boldsymbol{MFP}$)
3. Inflationary Fixed Point Logic ($\boldsymbol{IFP}$)
4. Partial Fixed Point Logic ($\boldsymbol{PFP}$)

In our presentation we plane to prove that $\boldsymbol{LFP}$ and $\boldsymbol{MFP}$ are institutions.

References

1. Rod Burstall and Joseph Goguen. The semantics of Clear, a specification language 1979 Copen Winter School on Abstract Software Specification volume 86 of Lectures Notes in Computer Science, pages 292-332. Springer 1980.
2. Joseph Goguen and Rod Burstall. Introductory institutions Proceedings, Logics of Programming Workshop, volume 164 of Lecture Notes in Computer Science pages 221-256, 19
3. Joseph Goguen and Rod Burstall. Institutions: Abstract model theory for specification and programming. Journal of the Association for Computing Machinery 39 (1):95-146, 1992
4. Daniel Gaina and Andrei Popescu. An institution-independent generalization of Tarski's Elementary Chain Theorem Journal of Logic and Computation, 16(6):713-735, 2006
5. Daniel Gaina and Andrei Popescu. An intitution-independent proof of Robinson consistency theorem. Studia Logica, 85(1):41-73, 2007
6. Razvan Diaconescu. Institution-independent ultraproducts. *Fundameta Informatica*, 55(3-4):321-348
7. Razvan Diaconescu. Elementary diagrams in institutions, *Journal of Logic and Computation*, 14(5):651-674, 2004
8. Razvan Diaconescu. An institution-independent proof of Craig Interpolation Theorem. Studia Logica, 77(1):59-79, 2004
9. Razvan Diaconescu. Borrowing interpolation *Journal Logic Computation*22(3): 561-586 2012.
10. Marius Petria and Razvan Diaconescu. Abstract Beth definability in institutions. *Journal of Symbolic Logic*, 71(3):1002-1028,2006
11. Marius Petria. An institutional version of Godel Completeness Theorem. In Algebra and Coalgebra in Computer Science, volume 4624, pages 409-424, Springer Berlin/Heidelberg, 2007
12. Petros Stefaneas and Razvan Diaconescu. Ultraproducts and possible worlds semantics in institutions. Theoritical Computer Science, 379(1):210-230, 2007
13. Razvan Diaconescu, Joseph Goguen and Petros Stefaneas. Logical support for modularisation. Logical Environments pages 83-130, Cambridge 1993. Proceedins of a Workshop held in Edinburg, Scotland, May 1991.
14. Razvan Diaconescu. Institution-independent Model Theory Studies in Universal Logic Springe Birkhauser, 2008
15. G.S. Boolos, J.P. Burgess, R.C. Jeffrey Computability and Logic Cambridge Press 2007
16. Leonid Likin Elements of Finite Model Theory Springer 2012
17. Mihai Codescu and Daniel Gaina. Birkhoff completeness in institutions. Logica Universalis , 2(2):277 309, 2008

A SOC-Based Formal Specification and Verification of Hybrid Systems

Ning Yu and Martin Wirsing

Department of Computer Science, University of Munich
Oettingenstrasse 67, 80538 Munich, Germany
{yu,wirsing}@pst.ifi.lmu.de
http://www.pst.ifi.lmu.de/

Service-Oriented Computing (SOC) is a computing paradigm that utilizes services as fundamental elements to support rapid, low-cost development of distributed applications in heterogeneous environments[1]. In SOC, a service is defined as an independent and autonomous piece of functionality which can be described, published, discovered and used in a uniform way. Within the development of SOC, complex systems are more and more involved. A typical type of complex systems are the hybrid systems, which arise in embedded control where discrete components are coupled with continuous components. In an abstract point of view, hybrid systems are mixtures of real-time (continuous) dynamics and discrete events[2]. In order to address these two aspects into SOC paradigm, we make our approach by giving a SOC-based formal specification and verification to hybrid systems.

The SOC-based formal specification of hybrid systems are realized by giving a hybrid extension to the SENSORIA Reference Modeling Language (SRML)[8][9]. SRML is a modeling language that can address the higher levels of abstraction of "business modeling"[3], developed in the project SENSORIA – the IST-FET Integrated Project that develops methodologies and tools such as Web Services[4] for dealing with the challenges arose in Service-Oriented Computing. By defining the Hybrid Doubly Labeled Transition Systems (HL^2TSs) which extend the L^2TSs of the branching time logic UCTL[5], we give a semantic domain over which the SRML extension could be defined and interpreted. Then we extend SRML syntax with a set of differential equation-based expression and define the formal semantics. In the extension, we combine the branching time logic UCTL with the dynamic temporal logic dTL[6] as the logic basis for reasoning about properties of SOC-based hybrid systems, in that we redefined dTL formulas and interpret them over HL^2TSs.

We illustrate our approach of verification though a case study of the European Train Control System (ETCS)[7], which is a a signalling, control and train protection system designed to replace the many incompatible safety systems currently used by European railways. In such a system the displacement of the train is governed by ordinary differential equations. Besides specifying the system with extended SRML, we verify it's safety property with a set of sequent calculus provided in [6] for verifying hybrid systems in dTL.

References

1. Georgakopoulos, D., Papazoglou, M.: Service-Oriented Computing. The MIT Press Cambridge, Massachusetts (2009)
2. van der Schaft, A., Schumacher, H.: An Introduction to Hybrid Dynamical Systems. Springer London, UK (1999)
3. Abreu, J., Bocchi, L., Fiadeiro, J., Lopes, A.: Specifying and Coomposing Interaction Protocols for Service-Oriented System Modelling. In: Derrick J., Vain J. (eds.) FORTE 2007. LNCS, vol. 4574, pp. 358–373. Springer, Berlin Heidelberg (2007)
4. Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web Services. Springer, New York (2004)
5. Maurice, H., ter Beek, Fantechi, A, Gnesi, S., Mazzanti, F.: An Action/State-Based Model-Checking Approach for the Analysis of An Asynchronous Protocol for Service-Oriented Applications. In: Leue S., Merino P. (eds.) FMICS 2007. LNCS, vol. 4916, pp. 133–148. Springer, Heidelberg (2008)
6. Platzer A.: AVACS – Automatic Verification and Analysis of Complex Systems. Technical report No. 12, AVACS (2007)
7. European Train Control System (ETCS) Open Proofs – Open Source, `http://openetcs.org/`
8. Abreu, J.: Modelling Business Conversations in Service Component Architectures. PhD thesis, University of Leicester (2009)
9. Fiadeiro, J., Lopes, A.,Abreu, J.: A Formal Model for service-Oriented Interactions. In: Science of Computer Programming, vol. 77, pp. 577 – 608. Elsevier (2012)