Patricia M. Hill (Ed.)

LNCS 3901

Logic Based Program Synthesis and Transformation

15th International Symposium, LOPSTR 2005
London, UK, September 2005
Revised Selected Papers

Springer

TP311-53
L864
2005

Patricia M. Hill (Ed.)

Logic Based Program Synthesis and Transformation

15th International Symposium, LOPSTR 2005
London, UK, September 7-9, 2005
Revised Selected Papers

Springer

Volume Editor

Patricia M. Hill
University of Leeds, School of Computing
Leeds LS2 9JT, UK
E-mail: hill@comp.leeds.ac.uk

Library of Congress Control Number: 2006921342

CR Subject Classification (1998): F.3.1, D.1.1, D.1.6, D.2.4, I.2.2, F.4.1

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN	0302-9743
ISBN-10	3-540-32654-5 Springer Berlin Heidelberg New York
ISBN-13	978-3-540-32654-0 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11680093 06/3142 5 4 3 2 1 0

¥359.00元

Preface

This volume contains a selection of papers presented at LOPSTR 2005, the 15th International Symposium on Logic-Based Program Synthesis and Transformation, held September 7–9, 2005.

The aim of the LOPSTR series is to stimulate and promote international research and collaboration on logic-based program development. Previous LOPSTR events have been held in Manchester, UK (1991, 1992, 1998), Louvain-la-Neuve, Belgium (1993), Pisa, Italy (1994), Arnhem, The Netherlands (1995), Stockholm, Sweden (1996), Leuven, Belgium (1997), Venice, Italy (1999), London, UK (2000), Paphos, Cyprus (2001), Madrid, Spain (2002), Uppsala, Sweden (2003), Verona, Italy (2004). Since 1994 the proceedings have been published in the LNCS series of Springer.

We would like to thank all those who submitted papers to LOPSTR. Overall, we received 33 submissions (full papers and extended abstracts). Each submission was reviewed by at least three people. The committee decided to accept 17 of these papers for presentation and for inclusion in the pre-conference proceedings. This volume contains a selection of revised full versions of ten of these papers. Thanks to all the authors of the accepted papers for the versions printed here and their presentations of these papers at LOPSTR 2005. We would like to thank François Fages for agreeing to give an invited talk and his contribution of a short paper included in these proceedings.

I am very grateful to the Program Committee as well as all the external reviewers for the reviewing of the submitted papers and invaluable help in the selection of these papers for presentation.

The submission, reviewing, electronic Program Committee meeting and preparation of the pre-conference proceedings and these proceedings were greatly simplified by the use of EasyChair (see http://www.easychair.org/). Special thanks are therefore due to Andrei Voronkov, who developed and supports this system.

LOPSTR 2005 was held concurrently with SAS 2005, the Symposium on Static Analysis in Imperial College, University of London. I would like to thank the SAS 2005 organizers, and, particularly, Chris Hankin, who took on all the hard work of the overall planning of the events.

LOPSTR 2005 was sponsored by ALP, the Association for Logic Programming.

December 2005 Patricia M. Hill

Conference Organization

Program Chair

Patricia M. Hill

Program Committee

Maria Alpuente
Roberto Bagnara
Gilles Barthe
Annalisa Bossi
Giorgio Delzanno
John Gallagher
Lindsay Groves
Gopal Gupta
Michael Hanus
Michael Leuschel
Fabio Martinelli
Fred Mesnard
Andreas Podelski
Maurizio Proietti
German Puebla
C.R. Ramakrishnan
Abhik Roychoudhury
Wim Vanhoof

External Reviewers

Christel Baier
Stephen-John Craig
Julien Forest
Frank Huch
Andy King
Jim Lipton
Sun Meng
Tamara Rezk
Josep Silva

Jesús Correas
Vicent Estruch-Gregori
Angel Herranz
Siau-Cheng Khoo
Gabriele Lenzini
Salvador Lucas
Alberto Pettorossi
Jaime Sánchez-Hernández
Fausto Spoto

Table of Contents

Temporal Logic Constraints in the Biochemical Abstract Machine BIOCHAM

François Fages

INRIA Rocquencourt, France
Francois.Fages@inria.fr

Abstract. Recent progress in Biology and data-production technologies push research toward a new interdisciplinary field, named Systems Biology, where the challenge is to break the complexity walls for reasoning about large biomolecular interaction systems. Pioneered by Regev, Silverman and Shapiro, the application of process calculi to the description of biological processes has been a source of inspiration for many researchers coming from the programming language community.

In this presentation, we give an overview of the Biochemical Abstract Machine (BIOCHAM), in which biochemical systems are modeled using a simple language of reaction rules, and the biological properties of the system, known from experiments, are formalized in temporal logic. In this setting, the biological validation of a model can be done by model-checking, both qualitatively and quantitatively. Moreover, the temporal properties can be turned into specifications for learning modifications or refinements of the model, when incorporating new biological knowledge.

1 Introduction

Systems biology is a cross-disciplinary domain involving biology, computer science, mathematics, and physics, aiming at elucidating the high-level functions of the cell from their biochemical bases at the molecular level. At the end of the Nineties, research in Bioinformatics evolved, passing from the analysis of the genomic sequence to the analysis of post-genomic data and interaction networks (expression of RNA and proteins, protein-protein interactions, etc). The complexity of these networks requires a large research effort to develop symbolic notations and analysis tools applicable to biological processes and data.

Our objective with the design of the Biochemical Abstract Machine BIOCHAM [1, 2] is to offer a software environment for modeling complex cell processes, making simulations (i.e. "*In silico* experiments"), formalizing the biological properties of the system known from real experiments, checking them and using them as specification when refining a model. The most original aspect of our approach can be summarized by the following identifications:

biological model = transition system,
biological property = temporal logic formula,
biological validation = model-checking.

2 Syntax of Biomolecular Interaction Rules

The objects manipulated in BIOCHAM represent molecular compounds, ranging from small molecules to proteins and genes. The syntax of objects and reaction rules is given by the following grammar:

```
object   = molecule | molecule : : location
molecule = name | molecule-molecule |molecule~{name,...,name}
reaction = solution => solution | kinetics for solution => solution
solution = _ | object | number*object | solution+solution
```

The objects can be localized in space with the operator "::" followed by a location name, such as the membrane, the cytoplasm, the nucleus, etc. The binding operator - is used to represent the binding of a molecule on a gene, the complexation of two proteins, and any form of intermolecular bindings. The alteration operator "$\sim$" is used to attach a set of modifications to a protein, like for instance the set of its phosphorylated sites (as long as they impact its activity).

Reaction rules express elementary biochemical interactions. There are essentially seven main rule schemas :

- `G => G + A` for the synthesis of `A` by gene `G`,
- `A => _` for the degradation of `A`,
- `A + B => A-B` for the complexation of two proteins `A` and `B`,
- `A-B => A + B` for the reversed decomplexation,
- `A + B => A~{p} + B` for the phosphorylation of protein `A` at site *p* catalyzed by `B`,
- `A~{p} + B => A + B` for the reversed dephosphorylation,
- `A::L => A::L'` for the transport of `A` from location L to L'.

The reaction rules can also be given with a kinetic expression, like for instance `0.1*[A][B] for A + B => A-B` where a mass action law kinetics with constant rate `0.1` is specified for the formation of the complex.

This rule-based language is used to model biochemical systems at three abstraction levels which correspond to three formal semantics: boolean, concentration (continuous dynamics) and population (stochastic dynamics).

A second language based on Temporal Logic [3] is used in BIOCHAM to formalize the biological properties of the system, and validate a model by model-checking [4, 5]. More precisely, symbolic and numerical model-checking tools are used respectively for CTL in the boolean semantics, for LTL with constraints over real numbers in the concentration semantics, and for PCTL with constraints over integers in the stochastic semantics.

3 Boolean Semantics

The most abstract semantics is the boolean semantics which ignores kinetic expressions. In that semantics, a boolean variable is associated to each BIOCHAM object, representing simply its presence or absence in the system. Reaction rules are then interpreted as an *asynchronous transition system* over states defined by the vector of boolean variables (similarly to the term rewriting formalism used in [6]). A rule such as `A + B => C + D` defines four possible state transitions corresponding to the possible consumption of the reactants: $A \wedge B \rightarrow A \wedge B \wedge C \wedge D$, $A \wedge B \rightarrow \neg A \wedge B \wedge C \wedge D$, $A \wedge B \rightarrow A \wedge \neg B \wedge C \wedge D$, $A \wedge B \rightarrow \neg A \wedge \neg B \wedge C \wedge D$. In that semantics, the choice of asynchrony and non-determinism is important to represent basic biological phenomena such as competitive inhibition, where a reaction "hides" another one because it consumes the reactants before the other reaction can occur. Formally, the boolean semantics of a set of BIOCHAM rules is defined by a *Kripke structure* $K = (S, R)$ where S is the set of states defined by the vector of boolean variables, and $R \subseteq S \times S$ is the transition relation between states.

In that boolean semantics, Computation Tree Logic (CTL) formulae are used to formalize the known biological properties of the system, and to query such properties in a model. Given an initial state specifying the biological conditions of the property, typical CTL formulae used in this context are :

- $EF(P)$, abbreviated as `reachable(P)`, stating that the organism is able to produce molecule P;
- $\neg E(\neg Q\ U\ P)$, abbreviated as `checkpoint(Q,P)`, stating that Q is a checkpoint for producing P;
- $EG(P)$, abbreviated as `steady(P)`, stating that the system can remain infinitely in a set of states described by formula P;
- $AG(P)$, abbreviated as `stable(P)`, stating that the system remains infinitely in P and cannot escape;
- $AG((P \Rightarrow EF\ \neg P) \wedge (\neg P \Rightarrow EF\ P))$, abbreviated as `oscil(P)`, a necessary (yet not sufficient without strong fairness assumption) consition for oscillations w.r.t. the presence of molecule P;
- $AG((P \Rightarrow EF\ Q) \wedge (Q \Rightarrow EF\ P))$, abbreviated as `loop(P,Q)`, a necessary condition for the alternance between states P and Q.

BIOCHAM evaluates CTL properties through an interface to the OBDD-based symbolic model checker NuSMV [7]. This technology makes it possible to check or query large models, like the model of the cell cycle control involving 165 proteins and genes, 500 variables and 800 reaction rules reported in [5].

4 Concentration Semantics

Basically the same scheme is applied to quantitative models, where each rule is given with a kinetic expression. The concentration semantics associates to each BIOCHAM object a real number representing its concentration. A set of BIOCHAM reaction rules $E = \{e_i \text{ for } S_i \Rightarrow S'_i\}_{i=1,\dots,n}$ with variables $\{x_1, \dots, x_m\}$, is then interpreted by the following set of (non-linear) ordinary differential equations (ODE) :

$$dx_k/dt = \sum_{i=1}^{n} r_i(x_k) * e_i - \sum_{j=1}^{n} l_j(x_k) * e_j$$

where $r_i(x_k)$ (resp. l_i) is the stoichiometric coefficient of x_k in the right (resp. left) member of rule i. Given an initial state, i.e. initial concentrations for each of the objects, the evolution of the system is deterministic and numerical integration methods compute discrete time series (i.e. linear Kripke structures) describing the evolution of the concentrations over time.

The concentration semantics being deterministic, Linear Time Logic (LTL) is used here to formalize the temporal properties. A first-order fragment of LTL is used to express numerical constraints on the concentrations of the molecules, or on their derivatives. For instance, `F([A]>10)` expresses that the concentration of `A` eventually gets above the threshold value 10. Oscillation properties, abbreviated as `oscil(M,K)`, are defined here as a change of sign of the derivative of M at least K times. These LTL formulae with constraints are checked with an ad-hoc model-checker implemented in Prolog, using the trace of the numerical integration of the ODEs associated to the rules.

5 Population Semantics

The population semantics is the most realistic semantics. It associates to each BIOCHAM object an integer representing the number of molecules in the system, and interprets reaction rules as a continuous time Markov chain. The kinetic expression e_i for the reaction i is converted into a transition rate τ_i (giving a transition probability after normalization) as follows [8]:

$$\tau_i = e_i \times (V_i \times K)^{(1-\sum_{k=1}^{m} l_i(x_k))} \times \prod_{k=1}^{m}(!l_i(x_k))$$

where l_i is the stoichiometric coefficient of the reactant x_k in the reaction rule i. Stochastic simulation techniques [9] compute realizations of the process. They are generally noisy versions of those obtained with the concentration semantics, however qualitatively different behaviors may also appear when small number of molecules are considered, which justifies the use of a stochastic dynamics.

In this setting, LTL formulae can be evaluated with their probability using a Monte Carlo method, which has proved to be more efficient than existing model-checkers for the probabilistic temporal logic PCTL. However, both the stochastic simulation and the model-checking are computationally more expensive than in the concentration semantics.

6 Learning Reaction Rules from Temporal Properties

Beyond making simulations, and checking properties of the models, the temporal properties can also be turned into specifications and temporal logic constraints for automatically searching and learning modifications or refinements of the model, when incorporating new biological knowledge. This is implemented in BIOCHAM by a combination of model-checking and search in the three abstraction levels.

This methodology is currently investigated with models of the cell cycle control (which regulates cell division) for the learning of kinetic parameter values from LTL properties in the concentration semantics [10], and for the learning of reaction rules from CTL properties in the boolean semantics [11]. A coupled model of the cell cycle and the circadian cycle is under development along these lines in BIOCHAM with applications to cancer chronotherapies.

Acknowledgements. This is a joint work with Nathalie Chabrier-Rivier, Sylvain Soliman and Laurence Calzone, with contributions from Sakina Ayata, Loïc Fosse, Lucie Gentils, Shrivaths Rajagopalan and Nathalie Sznajder. Support and fruitful discussions with our partners of the EU STREP project April-II are warmly acknowledged.

References

1. Fages, F., Soliman, S., Chabrier-Rivier, N.: Modelling and querying interaction networks in the biochemical abstract machine BIOCHAM. Journal of Biological Physics and Chemistry **4** (2004) 64–73
2. Chabrier, N., Fages, F., Soliman, S.: BIOCHAM's user manual. INRIA. (2003–2005)
3. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (1999)
4. Chabrier, N., Fages, F.: Symbolic model cheking of biochemical networks. In Priami, C., ed.: CMSB'03: Proceedings of the first Workshop on Computational Methods in Systems Biology. Volume 2602 of Lecture Notes in Computer Science., Rovereto, Italy, Springer-Verlag (2003) 149–162
5. Chabrier-Rivier, N., Chiaverini, M., Danos, V., Fages, F., Schächter, V.: Modeling and querying biochemical interaction networks. Theoretical Computer Science **325** (2004) 25–44
6. Eker, S., Knapp, M., Laderoute, K., Lincoln, P., Meseguer, J., Sönmez, M.K.: Pathway logic: Symbolic analysis of biological signaling. In: Proceedings of the seventh Pacific Symposium on Biocomputing. (2002) 400–412
7. Cimatti, A., Clarke, E., Enrico Giunchiglia, F.G., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: Nusmv 2: An opensource tool for symbolic model checking. In: Proceedings of the International Conference on Computer-Aided Verification, CAV'02, Copenhagen, Danmark (2002)
8. Gibson, M.A., Bruck, J.: A probabilistic model of a prokaryotic gene and its regulation. In Bolouri, H., Bower, J., eds.: Computational Methods in Molecular Biology: From Genotype to Phenotype. MIT press (2000)
9. Gillespie, D.T.: General method for numerically simulating stochastic time evolution of coupled chemical-reactions. Journal of Computational Physics **22** (1976) 403–434
10. Calzone, L., Chabrier-Rivier, N., Fages, F., Soliman, S.: A machine learning approach to biochemical reaction rules discovery. In III, F.J.D., ed.: Proceedings of Foundations of Systems Biology and Engineering FOSBE'05, Santa Barbara (2005) 375–379
11. Calzone, L., Chabrier-Rivier, N., Fages, F., Gentils, L., Soliman, S.: Machine learning bio-molecular interactions from temporal logic properties. In Plotkin, G., ed.: CMSB'05: Proceedings of the third Workshop on Computational Methods in Systems Biology. (2005)

Declarative Programming with Function Patterns*

Sergio Antoy[1] and Michael Hanus[2]

[1] Computer Science Dept., Portland State University, Oregon, USA
antoy@cs.pdx.edu
[2] Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany
mh@informatik.uni-kiel.de

Abstract. We propose an extension of functional logic languages that allows the definition of operations with patterns containing other defined operation symbols. Such "function patterns" have many advantages over traditional constructor patterns. They allow a direct representation of specifications as declarative programs, provide better abstractions of patterns as first-class objects, and support the high-level programming of queries and transformation of complex structures. Moreover, they avoid known problems that occur in traditional programs using strict equality. We define their semantics via a transformation into standard functional logic programs. Since this transformation might introduce an infinite number of rules, we suggest an implementation that can be easily integrated with existing functional logic programming systems.

1 Motivation

Functional logic languages (see [16] for a survey) integrate the most important features of functional and logic languages to provide a variety of programming concepts to the programmer. For instance, the concepts of demand-driven evaluation, higher-order functions, and polymorphic typing from functional programming are combined with logic programming features like computing with partial information (logic variables), constraint solving, and non-deterministic search for solutions. This combination, supported by optimal evaluation strategies [6] and new design patterns [8], leads to better abstractions in application programs such as implementing graphical user interfaces [18] or programming dynamic web pages [19].

A functional logic program consists of a set of datatype definitions and a set of functions or operations, defined by equations or rules, that operate on these types. For instance, the concatenation operation "++" on lists can be defined by the following two rules, where "[]" denotes the empty list and "x:xs" the non-empty list with first element x and tail xs:

```
[]     ++ ys = ys
(x:xs) ++ ys = x : xs++ys
```

Expressions are evaluated by rewriting with rules of this kind. For instance, [1,2] ++[3] evaluates to [1,2,3], where [x_1,x_2,...,x_n] denotes x_1:x_2:...:x_n:[], in three rewrite steps:

* This work was partially supported by the German Research Council (DFG) under grant Ha 2457/5-1 and the NSF under grant CCR-0218224.

```
[1,2]++[3]  →  1:([2]++[3])  →  1:(2:([]++[3]))  →  [1,2,3]
```

Beyond such functional-like evaluations, functional logic languages also compute with unknowns (logic variables). For instance, a functional logic language is able to *solve* an equation like `xs++[x] =:= [1,2,3]` (where `xs` and `x` are logic variables) by guessing the bindings `[1,2]` and 3 for `xs` and `x`, respectively.

This constraint solving capability can be exploited to define new operations using already defined functions. For instance, the operation `last`, which yields the last element of a list, can be defined as follows (the "`where...free`" clause declares logic variables in rules):

```
last l | xs++[x]=:=l = x   where xs,x free                    (last1)
```

In general, a *conditional equation* has the form l | $c = r$ and is applicable for rewriting if its condition c has been solved. A subtle point is the meaning of the symbol "`=:=`" used to denote equational constraints. Since modern functional logic languages, like Curry [17, 22] or Toy [25], are based on a non-strict semantics [6, 14] that supports lazy evaluation and infinite structures, it is challenging to compare arbitrary, in particular infinite, objects. Thus, the equality symbol "`=:=`" in a condition is usually interpreted as *strict equality*—the equation t_1 `=:=` t_2 is satisfied iff t_1 and t_2 are reducible to the same *constructor term* (see [13] for a more detailed discussion on this topic). A constructor term is a fully evaluated expression; a formal definition appears in Section 3.

Strict equality evaluates both its operands to a constructor term to prove the validity of the condition. For this reason, the strict equation "`x =:= head []`" does not hold for any `x`. The operation `head` is defined by the single rule `head (x:xs) = x`. Therefore, the evaluation of `head []` fails to obtain a constructor term. While the behavior of "`=:=`" is natural and intuitive in this example, it is less so in the following example.

A consequence of the strict equality in the definition of `last` in Display (*last1*) is that the list argument of `last` is fully evaluated. In particular, `last [failed,2]`, where `failed` is an operation whose evaluation fails, has no result. This outcome is unnatural and counterintuitive. In fact, the usual functional recursive definition of `last` would produce the expected result, 2, for the same argument. Thus, strict equality is harmful in this example (further examples will be shown later) since it evaluates more than one intuitively requires and, thus, reduces the inherent laziness of the computation.

There are good reasons for the usual definition of strict equality [13]; we will see that just dropping the strictness requirements in equational conditions leads to a non-intuitive behavior. Therefore, we propose in this paper an extension of functional logic languages with a new concept that solves all these problems: *function patterns*. Traditional patterns (i.e., the arguments of the left-hand sides of rules) are required to be constructor terms. Function patterns can also contain defined operation symbols so that the operation `last` is simply defined as

```
last (xs++[x]) = x
```

This definition leads not only to concise specifications, but also to a "lazier" behavior. Since the pattern variables `xs` and `x` are matched against the actual (possibly unevaluated) parameters, with this new definition of `last`, the expression `last [failed,2]` evaluates to 2.