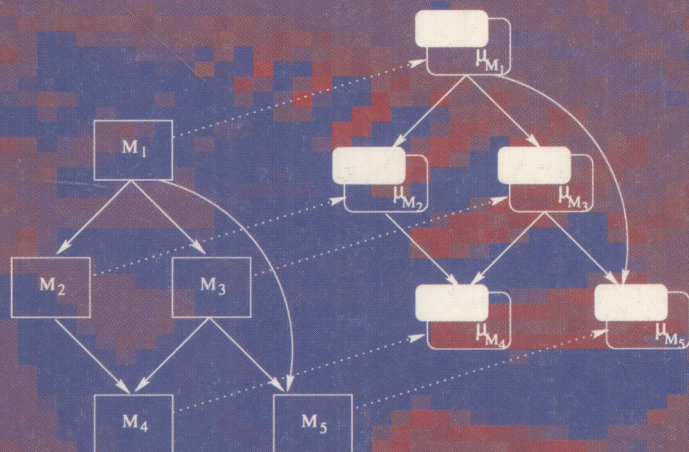


Maurice Bruynooghe  
Kung-Kiu Lau (Eds.)

# Program Development in Computational Logic

A Decade of Research Advances  
in Logic-Based Program Development



Springer

TP211.1-53  
1994  
Maurice Bruynooghe Kung-Kiu Lau (Eds.)

# Program Development in Computational Logic

A Decade of Research Advances  
in Logic-Based Program Development



E200404113



Springer

## Volume Editors

Maurice Bruynooghe  
Katholieke Universiteit Leuven  
Department of Computer Science  
Celestijnenlaan 200A, 3001 Heverlee, Belgium  
E-mail: Maurice.Bruynooghe@cs.kuleuven.ac.be

Kung-Kiu Lau  
University of Manchester  
Department of Computer Science  
Manchester M13 9PL, United Kingdom  
E-mail: kung-kiu@cs.man.ac.uk

Library of Congress Control Number: 2004107507

CR Subject Classification (1998): F.3.1, D.1.1, D.1.6, I.2.2, F.4.1, D.2, D.3

ISSN 0302-9743

ISBN 3-540-22152-2 Springer-Verlag Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable to prosecution under the German Copyright Law.

Springer-Verlag is a part of Springer Science+Business Media  
springeronline.com

© Springer-Verlag Berlin Heidelberg 2004  
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Boller Mediendesign  
Printed on acid-free paper SPIN: 11012061 06/3142 5 4 3 2 1 0

*Commenced Publication in 1973*

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

## Editorial Board

Takeo Kanade

*Carnegie Mellon University, Pittsburgh, PA, USA*

Josef Kittler

*University of Surrey, Guildford, UK*

Jon M. Kleinberg

*Cornell University, Ithaca, NY, USA*

Friedemann Mattern

*ETH Zurich, Switzerland*

John C. Mitchell

*Stanford University, CA, USA*

Moni Naor

*Weizmann Institute of Science, Rehovot, Israel*

Oscar Nierstrasz

*University of Bern, Switzerland*

C. Pandu Rangan

*Indian Institute of Technology, Madras, India*

Bernhard Steffen

*University of Dortmund, Germany*

Madhu Sudan

*Massachusetts Institute of Technology, MA, USA*

Demetri Terzopoulos

*New York University, NY, USA*

Doug Tygar

*University of California, Berkeley, CA, USA*

Moshe Y. Vardi

*Rice University, Houston, TX, USA*

Gerhard Weikum

*Max-Planck Institute of Computer Science, Saarbruecken, Germany*

**Springer**

*Berlin*

*Heidelberg*

*New York*

*Hong Kong*

*London*

*Milan*

*Paris*

*Tokyo*

# Preface

The tenth anniversary of the LOPSTR<sup>1</sup> symposium provided the incentive for this volume. LOPSTR started in 1991 as a workshop on logic program synthesis and transformation, but later it broadened its scope to logic-based program development in general, that is, program development in computational logic, and hence the title of this volume.

The motivating force behind LOPSTR has been the belief that declarative paradigms such as logic programming are better suited to program development tasks than traditional non-declarative ones such as the imperative paradigm. Specification, synthesis, transformation or specialization, analysis, debugging and verification can all be given logical foundations, thus providing a unifying framework for the whole development process.

In the past 10 years or so, such a theoretical framework has indeed begun to emerge. Even tools have been implemented for analysis, verification and specialization.

However, it is fair to say that so far the focus has largely been on programming-in-the-small. So the future challenge is to apply or extend these techniques to programming-in-the-large, in order to tackle software engineering in the real world.

Returning to this volume, our aim is to present a collection of papers that reflect significant research efforts over the past 10 years. These papers cover the whole development process: specification, synthesis, analysis, transformation and specialization, as well as semantics and systems.

We would like to thank all the authors for their valuable contributions that made this volume possible. We also thank the reviewers for performing their arduous task meticulously and professionally: Annalisa Bossi, Nicoletta Cocco, Bart Demoen, Danny De Schreye, Yves Deville, Sandro Etalle, Pierre Flener, John Gallagher, Samir Genaim, Gopal Gupta, Ian Hayes, Patricia Hill, Andy King, Vitaly Lagoon, Michael Leuschel, Naomi Lindenstrauss, Nancy Mazur, Mario Ornaghi, Dino Pedreschi, Alberto Pettorossi, Maurizio Proietti, CR Ramakrishnan, Sabina Rossi, Abhik Roychoudhury, Salvatore Ruggieri, Tom Schrijvers, Alexander Serebrenik, Jan-Georg Smaus, Wim Vanhoof and Sofie Verbaeten.

April 2004

Maurice Bruynooghe and Kung-Kiu Lau

---

<sup>1</sup> <http://www.cs.man.ac.uk/~kung-kiu/lopstr/>





# Lecture Notes in Computer Science

For information about Vols. 1–2997

please contact your bookseller or Springer-Verlag

- Vol. 3096: G. Melnik, H. Holz (Eds.), *Advances in Learning Software Organizations*. X, 173 pages. 2004.
- Vol. 3094: A. Nürnberger, M. Detyniecki (Eds.), *Adaptive Multimedia Retrieval*. VIII, 229 pages. 2004.
- Vol. 3093: S.K. Katsikas, S. Gritzalis, J. Lopez (Eds.), *Public Key Infrastructure*. XIII, 380 pages. 2004.
- Vol. 3092: J. Eckstein, H. Baumeister (Eds.), *Extreme Programming and Agile Processes in Software Engineering*. XVI, 358 pages. 2004.
- Vol. 3091: V. van Oostrom (Ed.), *Rewriting Techniques and Applications*. X, 313 pages. 2004.
- Vol. 3089: M. Jakobsson, M. Yung, J. Zhou (Eds.), *Applied Cryptography and Network Security*. XIV, 510 pages. 2004.
- Vol. 3086: M. Odersky (Ed.), *ECOOP 2004 – Object-Oriented Programming*. XIII, 611 pages. 2004.
- Vol. 3085: S. Berardi, M. Coppo, F. Damiani (Eds.), *Types for Proofs and Programs*. X, 409 pages. 2004.
- Vol. 3084: A. Persson, J. Stirna (Eds.), *Advanced Information Systems Engineering*. XIV, 596 pages. 2004.
- Vol. 3083: W. Emmerich, A.L. Wolf (Eds.), *Component Deployment*. X, 249 pages. 2004.
- Vol. 3078: S. Cotin, D.N. Metaxas (Eds.), *Medical Simulation*. XVI, 296 pages. 2004.
- Vol. 3077: F. Roli, J. Kittler, T. Windeatt (Eds.), *Multiple Classifier Systems*. XII, 386 pages. 2004.
- Vol. 3076: D. Buell (Ed.), *Algorithmic Number Theory*. XI, 451 pages. 2004.
- Vol. 3074: B. Kuijpers, P. Revesz (Eds.), *Constraint Databases and Applications*. XII, 181 pages. 2004.
- Vol. 3073: H. Chen, R. Moore, D.D. Zeng, J. Leavitt (Eds.), *Intelligence and Security Informatics*. XV, 536 pages. 2004.
- Vol. 3072: D. Zhang, A.K. Jain (Eds.), *Biometric Authentication*. XVII, 800 pages. 2004.
- Vol. 3070: L. Rutkowski, J. Siekmann, R. Tadeusiewicz, L.A. Zadeh (Eds.), *Artificial Intelligence and Soft Computing - ICAISC 2004*. XXV, 1208 pages. 2004. (Subseries LNAI).
- Vol. 3068: E. André, L. Dybkj\ae r, W. Minker, P. Heisterkamp (Eds.), *Affective Dialogue Systems*. XII, 324 pages. 2004. (Subseries LNAI).
- Vol. 3067: M. Dastani, J. Dix, A. El Fallah-Seghrouchni (Eds.), *Programming Multi-Agent Systems*. X, 221 pages. 2004. (Subseries LNAI).
- Vol. 3066: S. Tsumoto, R. Słowiński, J. Komorowski, J.W. Grzymala-Busse (Eds.), *Rough Sets and Current Trends in Computing*. XX, 853 pages. 2004. (Subseries LNAI).
- Vol. 3065: A. Lomuscio, D. Nute (Eds.), *Deontic Logic in Computer Science*. X, 275 pages. 2004. (Subseries LNAI).
- Vol. 3064: D. Bienstock, G. Nemhauser (Eds.), *Integer Programming and Combinatorial Optimization*. XI, 445 pages. 2004.
- Vol. 3063: A. Llamas, A. Strohmeier (Eds.), *Reliable Software Technologies - Ada-Europe 2004*. XIII, 333 pages. 2004.
- Vol. 3062: J.L. Pfaltz, M. Nagl, B. Böhlen (Eds.), *Applications of Graph Transformations with Industrial Relevance*. XV, 500 pages. 2004.
- Vol. 3060: A.Y. Tawfik, S.D. Goodwin (Eds.), *Advances in Artificial Intelligence*. XIII, 582 pages. 2004. (Subseries LNAI).
- Vol. 3059: C.C. Ribeiro, S.L. Martins (Eds.), *Experimental and Efficient Algorithms*. X, 586 pages. 2004.
- Vol. 3058: N. Sebe, M.S. Lew, T.S. Huang (Eds.), *Computer Vision in Human-Computer Interaction*. X, 233 pages. 2004.
- Vol. 3057: B. Jayaraman (Ed.), *Practical Aspects of Declarative Languages*. VIII, 255 pages. 2004.
- Vol. 3056: H. Dai, R. Srikant, C. Zhang (Eds.), *Advances in Knowledge Discovery and Data Mining*. XIX, 713 pages. 2004. (Subseries LNAI).
- Vol. 3055: H. Christiansen, M.-S. Hacid, T. Andreassen, H.L. Larsen (Eds.), *Flexible Query Answering Systems*. X, 500 pages. 2004. (Subseries LNAI).
- Vol. 3054: I. Crnkovic, J.A. Stafford, H.W. Schmidt, K. Wallnau (Eds.), *Component-Based Software Engineering*. XI, 311 pages. 2004.
- Vol. 3053: C. Bussler, J. Davies, D. Fensel, R. Studer (Eds.), *The Semantic Web: Research and Applications*. XIII, 490 pages. 2004.
- Vol. 3052: W. Zimmermann, B. Thalheim (Eds.), *Abstract State Machines 2004. Advances in Theory and Practice*. XII, 235 pages. 2004.
- Vol. 3051: R. Berghammer, B. Möller, G. Struth (Eds.), *Relational and Kleene-Algebraic Methods in Computer Science*. X, 279 pages. 2004.
- Vol. 3050: J. Domingo-Ferrer, V. Torra (Eds.), *Privacy in Statistical Databases*. IX, 367 pages. 2004.
- Vol. 3049: M. Bruynooghe, K.-K. Lau (Eds.), *Program Development in Computational Logic*. VIII, 539 pages. 2004.
- Vol. 3047: F. Oquendo, B. Warboys, R. Morrison (Eds.), *Software Architecture*. X, 279 pages. 2004.
- Vol. 3046: A. Laganà, M.L. Gavrilova, V. Kumar, Y. Mun, C.K. Tan, O. Gervasi (Eds.), *Computational Science and Its Applications – ICCSA 2004*. LIII, 1016 pages. 2004.



- Vol. 3045: A. Laganà, M.L. Gavrilova, V. Kumar, Y. Mun, C.K. Tan, O. Gervasi (Eds.), Computational Science and Its Applications – ICCSA 2004. LIII, 1040 pages. 2004.
- Vol. 3044: A. Laganà, M.L. Gavrilova, V. Kumar, Y. Mun, C.K. Tan, O. Gervasi (Eds.), Computational Science and Its Applications – ICCSA 2004. LIII, 1140 pages. 2004.
- Vol. 3043: A. Laganà, M.L. Gavrilova, V. Kumar, Y. Mun, C.K. Tan, O. Gervasi (Eds.), Computational Science and Its Applications – ICCSA 2004. LIII, 1180 pages. 2004.
- Vol. 3042: N. Mitrou, K. Kontovasilis, G.N. Rouskas, I. Iliadis, L. Merakos (Eds.), NETWORKING 2004, Networking Technologies, Services, and Protocols; Performance of Computer and Communication Networks; Mobile and Wireless Communications. XXXIII, 1519 pages. 2004.
- Vol. 3040: R. Conejo, M. Urretavizcaya, J.-L. Pérez-de-la-Cruz (Eds.), Current Topics in Artificial Intelligence. XIV, 689 pages. 2004. (Subseries LNAI).
- Vol. 3039: M. Bubak, G.D.v. Albada, P.M. Sloot, J.J. Dongarra (Eds.), Computational Science - ICCS 2004. LXVI, 1271 pages. 2004.
- Vol. 3038: M. Bubak, G.D.v. Albada, P.M. Sloot, J.J. Dongarra (Eds.), Computational Science - ICCS 2004. LXVI, 1311 pages. 2004.
- Vol. 3037: M. Bubak, G.D.v. Albada, P.M. Sloot, J.J. Dongarra (Eds.), Computational Science - ICCS 2004. LXVI, 745 pages. 2004.
- Vol. 3036: M. Bubak, G.D.v. Albada, P.M. Sloot, J.J. Dongarra (Eds.), Computational Science - ICCS 2004. LXVI, 713 pages. 2004.
- Vol. 3035: M.A. Wimmer (Ed.), Knowledge Management in Electronic Government. XII, 326 pages. 2004. (Subseries LNAI).
- Vol. 3034: J. Favela, E. Menasalvas, E. Chávez (Eds.), Advances in Web Intelligence. XIII, 227 pages. 2004. (Subseries LNAI).
- Vol. 3033: M. Li, X.-H. Sun, Q. Deng, J. Ni (Eds.), Grid and Cooperative Computing. XXXVIII, 1076 pages. 2004.
- Vol. 3032: M. Li, X.-H. Sun, Q. Deng, J. Ni (Eds.), Grid and Cooperative Computing. XXXVII, 1112 pages. 2004.
- Vol. 3031: A. Butz, A. Krüger, P. Olivier (Eds.), Smart Graphics. X, 165 pages. 2004.
- Vol. 3030: P. Giorgini, B. Henderson-Sellers, M. Winikoff (Eds.), Agent-Oriented Information Systems. XIV, 207 pages. 2004. (Subseries LNAI).
- Vol. 3029: B. Orchard, C. Yang, M. Ali (Eds.), Innovations in Applied Artificial Intelligence. XXI, 1272 pages. 2004. (Subseries LNAI).
- Vol. 3028: D. Neuenschwander, Probabilistic and Statistical Methods in Cryptology. X, 158 pages. 2004.
- Vol. 3027: C. Cachin, J. Camenisch (Eds.), Advances in Cryptology - EUROCRYPT 2004. XI, 628 pages. 2004.
- Vol. 3026: C. Ramamoorthy, R. Lee, K.W. Lee (Eds.), Software Engineering Research and Applications. XV, 377 pages. 2004.
- Vol. 3025: G.A. Vouras, T. Panayiotopoulos (Eds.), Methods and Applications of Artificial Intelligence. XV, 546 pages. 2004. (Subseries LNAI).
- Vol. 3024: T. Pajdla, J. Matas (Eds.), Computer Vision - ECCV 2004. XXVIII, 621 pages. 2004.
- Vol. 3023: T. Pajdla, J. Matas (Eds.), Computer Vision - ECCV 2004. XXVIII, 611 pages. 2004.
- Vol. 3022: T. Pajdla, J. Matas (Eds.), Computer Vision - ECCV 2004. XXVIII, 621 pages. 2004.
- Vol. 3021: T. Pajdla, J. Matas (Eds.), Computer Vision - ECCV 2004. XXVIII, 633 pages. 2004.
- Vol. 3019: R. Wyrzykowski, J.J. Dongarra, M. Paprzycki, J. Wasniewski (Eds.), Parallel Processing and Applied Mathematics. XIX, 1174 pages. 2004.
- Vol. 3018: M. Bruynooghe (Ed.), Logic Based Program Synthesis and Transformation. X, 233 pages. 2004.
- Vol. 3016: C. Lengauer, D. Batory, C. Consel, M. Odersky (Eds.), Domain-Specific Program Generation. XII, 325 pages. 2004.
- Vol. 3015: C. Barakat, I. Pratt (Eds.), Passive and Active Network Measurement. XI, 300 pages. 2004.
- Vol. 3014: F. van der Linden (Ed.), Software Product-Family Engineering. IX, 486 pages. 2004.
- Vol. 3012: K. Kurumatani, S.-H. Chen, A. Ohuchi (Eds.), Multi-Agents for Mass User Support. X, 217 pages. 2004. (Subseries LNAI).
- Vol. 3011: J.-C. Régim, M. Rueher (Eds.), Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems. XI, 415 pages. 2004.
- Vol. 3010: K.R. Apt, F. Fages, F. Rossi, P. Szeredi, J. Vánca (Eds.), Recent Advances in Constraints. VIII, 285 pages. 2004. (Subseries LNAI).
- Vol. 3009: F. Bomarius, H. Iida (Eds.), Product Focused Software Process Improvement. XIV, 584 pages. 2004.
- Vol. 3008: S. Heuel, Uncertain Projective Geometry. XVII, 205 pages. 2004.
- Vol. 3007: J.X. Yu, X. Lin, H. Lu, Y. Zhang (Eds.), Advanced Web Technologies and Applications. XXII, 936 pages. 2004.
- Vol. 3006: M. Matsui, R. Zuccherato (Eds.), Selected Areas in Cryptography. XI, 361 pages. 2004.
- Vol. 3005: G.R. Raidl, S. Cagnoni, J. Branke, D.W. Corne, R. Drechsler, Y. Jin, C.G. Johnson, P. Machado, E. Marchiori, F. Rothlauf, G.D. Smith, G. Squillero (Eds.), Applications of Evolutionary Computing. XVII, 562 pages. 2004.
- Vol. 3004: J. Gottlieb, G.R. Raidl (Eds.), Evolutionary Computation in Combinatorial Optimization. X, 241 pages. 2004.
- Vol. 3003: M. Keijzer, U.-M. O'Reilly, S.M. Lucas, E. Costa, T. Soule (Eds.), Genetic Programming. XI, 410 pages. 2004.
- Vol. 3002: D.L. Hicks (Ed.), Metainformatics. X, 213 pages. 2004.
- Vol. 3001: A. Ferscha, F. Mattern (Eds.), Pervasive Computing. XVII, 358 pages. 2004.
- Vol. 2999: E.A. Boiten, J. Derrick, G. Smith (Eds.), Integrated Formal Methods. XI, 541 pages. 2004.
- Vol. 2998: Y. Kameyama, P.J. Stuckey (Eds.), Functional and Logic Programming. X, 307 pages. 2004.

# Table of Contents

## Specification and Synthesis

Specifying Compositional Units for Correct Program Development in Computational Logic .....	1
<i>Kung-Kiu Lau and Mario Ornaghi</i>	
Synthesis of Programs in Computational Logic .....	30
<i>David Basin, Yves Deville, Pierre Flener, Andreas Hamfelt, and Jørgen Fischer Nilsson</i>	
Developing Logic Programs from Specifications Using Stepwise Refinement .....	66
<i>Robert Colvin, Lindsay Groves, Ian J. Hayes, David Hemer, Ray Nickson, and Paul Strooper</i>	

## Semantics

Declarative Semantics of Input Consuming Logic Programs .....	90
<i>Annalisa Bossi, Nicoletta Cocco, Sandro Etalle, and Sabina Rossi</i>	
On the Semantics of Logic Program Composition .....	115
<i>Antonio Brogi</i>	

## Analysis

Analysing Logic Programs by Reasoning Backwards .....	152
<i>Jacob M. Howe, Andy King, and Lunjin Lu</i>	
Binding Time Analysis for Mercury .....	189
<i>Wim Vanhoof, Maurice Bruynooghe, and Michael Leuschel</i>	
A Generic Framework for Context-Sensitive Analysis of Modular Programs .....	233
<i>Germán Puebla, Jesús Correias, Manuel V. Hermenegildo, Francisco Bueno, María García de la Banda, Kim Marriott, and Peter J. Stuckey</i>	

## Transformation and Specialisation

Unfold/Fold Transformations for Automated Verification of Parameterized Concurrent Systems .....	261
<i>Abhik Roychoudhury and C.R. Ramakrishnan</i>	

Transformation Rules for Locally Stratified Constraint Logic Programs ..	291
<i>Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti</i>	
Specialising Interpreters Using Offline Partial Deduction .....	340
<i>Michael Leuschel, Stephen J. Craig, Maurice Bruynooghe, and Wim Vanhoof</i>	
<b>Termination</b>	
Characterisations of Termination in Logic Programming .....	376
<i>Dino Pedreschi, Salvatore Ruggieri, and Jan-Georg Smaus</i>	
On the Inference of Natural Level Mappings .....	432
<i>Jonathan C. Martin and Andy King</i>	
Proving Termination for Logic Programs by the Query-Mapping Pairs Approach .....	453
<i>Naomi Lindenstrauss, Yehoshua Sagiv, and Alexander Serebrenik</i>	
<b>Systems</b>	
Herbrand Constraint Solving in HAL .....	499
<i>Bart Demoen, María García de la Banda, Warwick Harvey, Kim Marriott, David Overton, and Peter J. Stuckey</i>	
<b>Author Index</b> .....	539

# Specifying Compositional Units for Correct Program Development in Computational Logic

Kung-Kiu Lau<sup>1</sup> and Mario Ornaghi<sup>2</sup>

<sup>1</sup> Department of Computer Science, University of Manchester  
Manchester M13 9PL, United Kingdom  
`kung-kiu@cs.man.ac.uk`

<sup>2</sup> Dipartimento di Scienze dell'Informazione, Università degli studi di Milano  
Via Comelico 39/41, 20135 Milano, Italy  
`ornaghi@dsi.unimi.it`

**Abstract.** In order to provide a formalism for defining program correctness and to reason about program development in Computational Logic, we believe that it is better to distinguish between specifications and programs. To this end, we have developed a general approach to specification that is based on a model-theoretic semantics. In our previous work, we have shown how to define specifications and program correctness for open logic programs. In particular we have defined a notion of correctness called *steadfastness*, that captures at once modularity, reusability and correctness. In this paper, we review our past work and we show how it can be used to define compositional units that can be correctly reused in modular or component-based software development.

## 1 Introduction

In software engineering, requirements analysis, design and implementation are distinctly separate phases of the development process [18], as they employ different methods and produce different artefacts. In requirements analysis and design, *specifications* play a central role, as a frame of reference capturing the requirements and the design decisions. By contrast, data and programs only appear in the implementation phase, towards the end of the development process. There is therefore a clear distinction between specifications and programs.

In Computational Logic, however, this distinction is usually not maintained. This is because there is a widely held view that logic programs are executable specifications and therefore there is no need to produce specifications before the implementation phase of the development process. We believe that undervaluing specifications in this manner is not an ideal platform for program development. If programs are indistinguishable from specifications, then how do we define program correctness, and how do we reason about program development? We hold the view that the meaning of correctness must be defined in terms of something other than logic programs themselves. We are not alone in this, see e.g., [17, p. 410]. In our view, the specification should axiomatise all our relevant knowledge of the problem context and the necessary data types, whereas,

for complexity reasons, programs rightly capture only what is strictly necessary for computing. In the process of extracting programs from specifications, a lot of knowledge is lost, making programs much weaker axiomatisations. This suggests that specifying and programming are different activities, involving different methodological aspects. Thus, we take the view that specifications should be clearly distinguished from programs, especially for the purpose of program development. Indeed, we have shown (in [28,29]) that in Computational Logic, not only can we maintain this distinction, but we can also define various kinds of specifications for different purposes. Moreover, we can also define correctness with respect to these specifications.

Our semantics for specification and correctness is model-theoretic. The declarative nature of such a semantics allows us to define *steadfastness* [34], a notion of correctness that captures at once modularity, reusability and correctness. *Open* programs are incomplete pieces of code that can be (re)used in many different *admissible* situations, by *closing* them (by adding the missing code) in many different ways. *Steadfastness* of an open program  $P$  is pre-proved correctness of the various closures of  $P$ , with respect to the different meanings that the specification of  $P$  assumes in the admissible situations. For correct reuse, we need to know when a situation is admissible. This knowledge is given by the *problem context*. We have formalised problem context as a *specification framework* [27], namely, a first-order theory that axiomatises the problem context, characterises the admissible situations as its (intended) models, and is used to write specifications and to reason about them.

In this paper, we review our work in *specification* and *correctness* of logic programs, including steadfastness. Our purpose is to discuss the role of steadfastness for correct software development. In particular, we are interested in modularity and reuse, which are key aspects of software development. Our work is centred on the notion of a *compositional unit*. A compositional unit is a *software component*, which is commonly defined as a unit of composition with contractually specified interfaces and context dependencies only [46]. The interfaces declare the imported and exported operations, and the context dependencies specify the constraints that must be satisfied in order to correctly (re)use them. Throughout the paper, we will not refer to compositional units as software components, however, for the simple reason that as yet there is no standard definition for the latter (although the one we used above [46] is widely accepted). So we prefer to avoid any unnecessary confusion. In our compositional units, the interfaces and the context dependencies are declaratively specified in the context of the specification framework  $\mathcal{F}$  axiomatising the problem context.  $\mathcal{F}$  gives a precise semantics to specifications and allows us to reason about the *correctness* of programs, as well as their *correct reuse*. Thus, in our formalisation, a compositional unit has a three-tier structure, with separate levels for framework, specifications and programs.

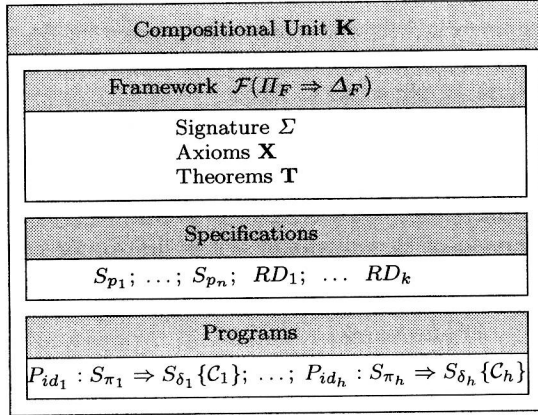
We introduce compositional units in Section 2, and consider the three levels separately. We focus on model-theoretic semantics of frameworks and specifications, and on steadfastness (i.e., open program correctness).

In Section 3, we show how the proposed formalisation of compositional units can be used to support correct reuse. Our aim is to highlight the aspects related to specifications, so we consider only the aspects related to the framework and the specification levels, while assuming the possibility of deriving (synthesising) steadfast programs from specifications.

At the end of each section we briefly discuss and compare our results with related work, and finally in the conclusion we comment on future developments.

## 2 Compositional Units

In our approach, compositional units represent correctly reusable units of *specifications and correct open programs*. Our view is that specifications and programs are not stand-alone entities, but are always to be considered in the light of a problem context. The latter plays a central role: it is the *semantic context* in which specifications and program correctness assume their appropriate meaning, and it contains the necessary knowledge for *reasoning* about correctness and correct reuse. This is reflected in the three-tier structure (with model-theoretic semantics) of a compositional unit, as illustrated in Figure 1.



**Fig. 1.** A three-tier formalism.

At the top level of a compositional unit  $K$ , we have a *specification framework*  $\mathcal{F}$ , or *framework* for short, that embodies an axiomatisation of the problem context.  $\mathcal{F}$  has a signature  $\Sigma$ , a set  $\mathbf{X}$  of *axioms*, a set  $\mathbf{T}$  of *theorems*, a list  $\Pi_F$  of *open symbols*, and a list  $\Delta_F$  of *defined symbols*. The syntax  $\Pi_F \Rightarrow \Delta_F$  indicates that the axioms of  $\mathcal{F}$  fix (the meaning of) the symbols  $\Delta_F$  whenever  $\mathcal{F}$  is composed with frameworks that fix  $\Pi_F$ . The defined and open symbols belong to the signature  $\Sigma$ , which may also contain *closed symbols*, namely symbols

defined completely by the axioms (i.e., independently from  $\Pi_F$ ). Frameworks are explained in Section 2.1, and framework composition is explained in Section 3.1.

In the middle, we have the *specification section*. Its role is to bridge the gap between the framework  $\mathcal{F}$  and the chosen *programming language*. So far, we have considered only logic programs, and the corresponding specification formalism is explained in Section 2.2. The specification section contains the specifications  $S_{p_1}, \dots, S_{p_n}$  of the program predicates occurring in the program section. It may also contain a set of *specification reduction theorems*  $RD_1, \dots, RD_k$ , that are useful to reason about correct reuse. Specification reduction is explained in Section 3.2.

At the bottom, we have the *program section*. Programs are open logic (or constraint logic) programs. An open program  $P_{id_i} : S_{\pi_i} \Rightarrow S_{\delta_i} \{C_i\}$  ( $1 \leq i \leq h$ ) has an identifier  $id_i$ , an *interface specification*  $S_{\pi_i} \Rightarrow S_{\delta_i}$  and a set  $\{C_i\}$  of *implementation clauses*.  $S_{\pi_i}$  and  $S_{\delta_i}$  are lists of specifications defined in the specification section. An interface specification contains all the information needed to *correctly reuse a correct* program. Programs and correctness are explained in Section 2.3. Correct reuse is explained in Section 3.3.

## 2.1 Specification Frameworks

A specification framework  $\mathcal{F}$  is defined in the context of first-order logic, and contains the *relevant knowledge* of the necessary concepts and data types for building a model of the application at hand.

We distinguish between *closed* and *open* frameworks. A *closed framework*  $\mathcal{F} = \langle \Sigma, \mathbf{X}, \mathbf{T} \rangle$  has a signature  $\Sigma$ , a set  $\mathbf{X}$  of axioms, and a set  $\mathbf{T}$  of theorems. It has no open and defined symbols, that is, all the symbols of  $\Sigma$  are closed.

*Example 1.* An example of closed framework is first-order arithmetic  $\mathcal{NAT} = \langle \Sigma_{Nat}, \mathbf{X}_{Nat}, \mathbf{T}_{Nat} \rangle$ , introduced by the following syntax:<sup>3</sup>

```

Framework  $\mathcal{NAT}$ ;
DECLS:    $Nat : sort$ ;
           $0 : [] \rightarrow Nat$ ;
           $s : [Nat] \rightarrow Nat$ ;
           $-+, -*_- : [Nat, Nat] \rightarrow Nat$ ;
AXS:    $Nat : construct(0, s : Nat)$ ;
         $+$  :  $i + 0 = i$ ;
            $i + s(j) = s(i + j)$ ;
         $:$     $i * 0 = 0$ ;
            $i * s(j) = i * j + i$ ;
THMS:   $i + j = j + i$ ;
        ...

```

<sup>3</sup> In all the examples, we will omit the outermost universal quantifiers, but their omnipresence should be implicitly understood.



The signature  $\Sigma_{Nat}$ , introduced in the declaration section `DECLS`, is the signature of Peano's arithmetic. The axioms  $\mathbf{X}_{Nat}$ , introduced in the `AXS` section, are the usual ones of first-order arithmetic. 0 and  $s$  are the *constructors* of  $Nat$  and their axioms, which we call the *constructor axioms* for  $Nat$ , are collectively indicated by *construct*(0,  $s : Nat$ ). The latter contains Clark's equality theory [35] for 0 and  $s$ , as well as all the instances of the first-order induction schema.  $\mathcal{NAT}$  has been widely studied, and there are a lot of known theorems (in section `THMS`), including for example the associative, commutative and distributive laws.

Theorems are an important part of a framework. However, they are not relevant in the definitions that follow, so we will not refer to them explicitly here.

For closed frameworks we adopt isoinitial semantics, that is, we choose the intended model of  $\mathcal{F} = \langle \Sigma, \mathbf{X} \rangle$  to be a *reachable isoinitial model*, defined as follows:

**Definition 1 (Reachable Isoinitial Model [5]).** *Let  $\mathbf{X}$  be a set of  $\Sigma$ -axioms. A  $\Sigma$ -structure  $\mathbf{I}$  is an isoinitial model of  $\mathbf{X}$  iff, for every model  $\mathbf{M}$  of  $\mathbf{X}$ , there is a unique isomorphic embedding  $i : \mathbf{I} \rightarrow \mathbf{M}$ .*

*A model  $\mathbf{I}$  is reachable if its elements can be represented by ground terms.*

**Definition 2 (Adequate Closed Frameworks [30]).** *A closed framework  $\mathcal{F} = \langle \Sigma, \mathbf{X} \rangle$  is adequate iff there is a reachable isoinitial model  $\mathbf{I}$  of  $\mathbf{X}$  that we call 'the' intended model of  $\mathcal{F}$ .*

In fact  $\mathbf{I}$  is one of many intended models of  $\mathcal{F}$ , all of which are isomorphic. So  $\mathbf{I}$  is unique up to isomorphism, and hence our (ab)use of 'the'.

As shown in [5], adequacy entails the computability of the operations and predicates of the signature.

*Example 2.*  $\mathcal{NAT}$  is an adequate closed framework. Its intended model is the standard structure  $\mathcal{N}$  of natural numbers ( $\mathcal{N}$  is a reachable isoinitial model of  $\mathbf{X}_{Nat}$ ).  $\mathcal{N}$  interprets  $Nat$  as the *set of natural numbers*, and  $s$ ,  $+$  and  $*$  as the *successor*, *sum* and *product* function, respectively.

The adequacy of a closed framework is not a decidable property. We have the following useful proof-theoretic characterisation, which can be seen as a "richness requirement" implicit in isoinitial semantics [31]:

**Definition 3 (Atomic Completeness).** *A framework  $\mathcal{F} = \langle \Sigma, \mathbf{X} \rangle$  is atomically complete iff, for every ground atomic formula  $A$ , either  $\mathbf{X} \vdash A$  or  $\mathbf{X} \vdash \neg A$ .*

**Theorem 1 (Adequacy Condition [38]).** *A closed framework  $\mathcal{F} = \langle \Sigma, \mathbf{X} \rangle$  is adequate iff it has at least one reachable model and is atomically complete.*

Closed adequate frameworks can be built incrementally, starting from a closed adequate kernel, by means of *adequate extensions*.

**Definition 4 (Adequate Extensions [30]).** An adequate extension of an adequate closed framework  $\mathcal{F} = \langle \Sigma, \mathbf{X} \rangle$  is an adequate closed framework  $\mathcal{F}_\delta = \langle \Sigma \cup \delta, \mathbf{X} \cup D_\delta \rangle$  such that:

- a)  $D_\delta$  is a set of  $(\Sigma \cup \delta)$ -axioms, axiomatising a set of new (i.e., not in  $\Sigma$ ) symbols  $\delta$ ;
- b) the  $\Sigma$ -reduct  $\mathbf{I}|_\Sigma$  of the intended model  $\mathbf{I}$  of  $\mathcal{F}_\delta$  is the intended model of  $\mathcal{F}$ .

The notions of *reduct* and *expansion* are standard in logic [4]. The  $\Sigma$ -reduct  $\mathbf{I}' = \mathbf{I}|_\Sigma$  forgets the interpretation of the symbols not in  $\Sigma$ , in our case the new symbols  $\delta$ . Conversely,  $\mathbf{I}$  is said to be a  $(\Sigma \cup \delta)$ -*expansion* of  $\mathbf{I}'$ , that is, a  $(\Sigma \cup \delta)$ -expansion is a  $(\Sigma \cup \delta)$ -interpretation that preserves the meaning of the old  $\Sigma$ -symbols, and interprets the new  $\delta$  arbitrarily.

In Definition 4, by b), the intended model  $\mathbf{I}$  of an adequate extension is an expansion of the old intended model, that is, adequacy entails that the meaning of the old symbols is preserved.

If the axioms  $D_\delta$  of an adequate extension are explicit definitions, we say that they are *adequate explicit definitions*. Since they are important in our approach, we briefly recall them.

An explicit definition of a new relation  $r$  has the form  $\forall \underline{x} \bullet r(\underline{x}) \leftrightarrow R(\underline{x})$ , where  $\underline{x}$  indicates a tuple of variables and (as usual) “ $\bullet$ ” extends the scope of a quantifier to the longest subformula next to it. The explicit definition of a new function  $f$  has the form  $\forall \underline{x} \bullet F(\underline{x}, f(\underline{x}))$ , where  $R(\underline{x})$  and  $F(\underline{x}, y)$  are formulas of the framework that contain free only the indicated variables. The explicit definition of  $f$  has the *proof obligation*  $\mathbf{X} \vdash \forall \underline{x} \bullet \exists! y \bullet F(\underline{x}, y)$ , where  $\mathbf{X}$  are the framework axioms (as usual,  $\exists! y$  means unique existence).  $R(\underline{x})$  is called the *definens* (or *defining formula*) of  $r$ , and  $F(\underline{x}, y)$  the *definiens* (or *defining formula*) of  $f$ .

Explicit definitions have nice properties. They are *purely* declarative, in the following sense: they define the new symbols *purely in terms of the old ones*, that is, in a *non-recursive* way. This declarative character is reflected by the following *eliminability* property, where  $\Sigma$  is the signature of the framework and  $\delta$  are the new explicitly defined symbols: the extension is conservative (i.e., no new  $\Sigma$ -theorem is added) and every formula of  $\Sigma + \delta$  is provably equivalent to a corresponding formula of the old signature  $\Sigma$ . Moreover, if we start from a sufficiently expressive kernel, most of the relevant relations and functions can be explicitly defined. Finally, we can prove:

**Proposition 1.** *If the definiens of an explicit definition is quantifier-free, then the definition is adequate.*

If the definiens is not quantifier-free, adequacy must be checked. To state the adequacy of closed frameworks and of explicit definitions, we can apply proof methods based on logic program synthesis [26,27] or constructive logic [38].

*Example 3.* The kernel  $\mathcal{NAT}$  of Example 1 is sufficiently expressive in the following sense. Every recursively enumerable relation  $r$  can be introduced by an