Maurice Bruynooghe   Jaan Penjam (Eds.)

Programming Language Implementation and Logic Programming

5th International Symposium, PLILP '93
Tallinn, Estonia, August 25-27, 1993
Proceedings

CC01-714

Springer-Verlag
Berlin Heidelberg New York
London Paris Tokyo
Hong Kong Barcelona
Budapest

Series Editors

Gerhard Goos
Universität Karlsruhe
Postfach 69 80
Vincenz-Priessnitz-Straße 1
D-76131 Karlsruhe, Germany

Juris Hartmanis
Cornell University
Department of Computer Science
4130 Upson Hall
Ithaca, NY 14853, USA

Volume Editors

Maurice Bruynooghe
Department of Computer Science, Katholieke Universiteit Leuven
Celestijnenlaan 200A, B-3001 Heverlee, Belgium

Jaan Penjam
Software Department, Institute of Cybernetics
Akadeemia tee 21, EE0026 Tallinn, Estonia

CR Subject Classification (1991): F.4.1-2, D.3.1, D.3.4, F.3.3, I.2.1

[illegible]

ISBN 3-540-57186-8 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-57186-8 Springer-Verlag New York Berlin Heidelberg

Typesetting: Camera-ready by authors
Printing and binding: Druckhaus Beltz, Hemsbach/Bergstr.
45/3140-543210 - Printed on acid-free paper

Preface

The international Symposium on Programming Language Implementation and Logic Programming (PLILP) is held every year. The series of PLILP symposiums was established by Pierre Deransart, Jan Małuszyński and Bernard Lorho with the aim to promote contacts and information exchange among scientists who share common interests in declarative programming techniques, logic programming and programming languages implementation. Researchers from the fields of algorithmic programming languages as well as logic, functional, object-oriented and constraint programming constitute the audience of PLILP.

This volume contains the papers which have been accepted for presentation at the Fifth International Symposium PLILP'93 held in Tallinn, Estonia, August 25-27, 1993. The preceding meetings took place in Orléans, France (May 16-18,1988), in Linköping, Sweden (August 20-22,1990), in Passau, Germany (August 26-28, 1991), and in Leuven, Belgium (August 26-28,1992), and their proceedings are published by Springer-Verlag as Lecture Notes of Computer Science, volumes 348, 456, 528 and 631 respectively. One of the goals of organizing PLILP'93 in Tallinn was to encourage scientific contacts between researchers from Eastern and Central European countries and the Western community of computer scientists.

In response to the call for papers 72 papers were submitted to the PLILP'93 by authors from all over the world. All submitted papers were reviewed by 2-4 experts. The program committee selected 24 papers on the basis of their scientific quality and relevance to the symposium. At the symposium, four invited talks were given by Alexander Dikovsky, Neil D. Jones, Uwe Kastens and Andrei Mantsivoda. Several software systems and poster presentations were presented, showing new developments in implementation of programming languages and declarative programming.

This volume contains three of the invited presentations, selected papers and abstracts of the selected system demonstrations.

On behalf of the program committee the program chairmen would like to thank all those who submitted papers and the people involved in the reviewing process. They are listed on the following pages.

The PLILP'93 will be hosted by the Institute of Cybernetics of the Estonian Academy of Sciences. The support of *Katholieke Universiteit Leuven, INRIA (Institut National de Recherche en Informatique et en Automatique), Estonian Informatics Fund* and *Siemens AG* is gratefully acknowledged.

We thank all who contributed to the Symposium and its organisation.

Tallinn - Leuven, M.Bruynooghe
June 1993 J.Penjam

Conference Chairmen

Jaan Penjam, Institute of Cybernetics, Tallinn (Estonia)
Maurice Bruynooghe, Katholieke Universiteit, Leuven (Belgium)

Program Committee

John Darlington, Imperial College, London (UK)
Saumya Debray, Univ. of Arizona, Tuscon (USA)
Wlodzimierz Drabent, Linköping Univ. (Sweden)
Gérard Ferrand, Université d'Orléans (France)
Manuel Hermenegildo, Technical Univ. of Madrid (Spain)
Bharat Jayaraman, State Univ. of New York, Buffalo (USA)
Feliks Kluzniak, Warsaw Univ. (Poland)
Brian Mayoh, Univ. of Åarhus (Denmark)
Alan Mycroft, Cambridge (UK)
Lee Naish, Univ. of Melbourne (Australia)
Jukka Paakki, Linköping Univ. (Sweden)
Peter Pepper, Technical Univ. of Berlin (Germany)
Igor Pottosin, Institute of Informatics Systems (Russia)
Antoine Rauzy, Laboratoire de Recherche en Informatique (France)
Jiro Tanaka, Fujitsi Laboratories, Tokyo (Japan)
Franco Turini, Universita di Pisa (Italy)
Andrey Voronkov, Uppsala Univ. (Sweden)
David Scott Warren, State Univ. of New York, Buffalo (USA)

Organizing Committee

Rein Lõugas, Institute of Cybernetics, Tallinn
Monika Perkmann, Institute of Cybernetics, Tallinn
Ahto Kalja, Institute of Cybernetics, Tallinn
Kaur Kiisler, Institute of Cybernetics, Tallinn
Anne Tavast, Institute of Cybernetics, Tallinn

List of Referees

Many other referees helped the Program Committee in evaluating papers. Their assistance is gratefully acknowledged.

Asperti, Andrea
Ballesteros, Francisco
Banda, Maria Jose G.
Barklund, Jonas
Bigot, Peter A.
Boye, Johan
Bueno, Francisco
Bulyonkov, Michail
Burn, Geoffrey
Cabeza, Daniel
Carro, Manuel
Chakravarty, Manuel
Chassin de Kergommeaux, J.
Clemente, Isabel G.
Clocksin, W. F.
Codish, Michael
Codognet, Christian
Codognet, Philippe
Coen, Martin
Corsini, M-M.
Dart, Philip
Davison, Andrew
De Schreye, Danny
Degerstedt, Lars
Demoen, Bart
Diaz, Daniel
Didrich, Klaus
Ducasse, M.
Evstigneev, Vladimir
Exner, Jürgen
Faulhaber, Joachim
Ferran, Guy
Garcia de la Banda,Maria
Garcia-Clemente,Isabel
Garcia-Martin, J.
Grieskamp, Wolfgang
Grudzinski, Grzegorz
Guo, Yike
Ida, T.
Janssens, Gerda
Jourdan, Martin
Kågedal, Andreas
Kasyanov, Victor
Kessler, Robert R.
Krepski, Artur
Launchbury, John
Lindstrom, Gary
Maeder, Christian
Maluszynski, Jan
Maričn, A.
Marino-Carballo, Julio
Matskin, Michail
Meriste, Merik
Moore, Marcus
Moreno-Navarro, J. J.
Mulkers, A
Nakagawa, Koji
Nesi, M.
Nielsen, Flemming
Nilsson, Ulf
Niwiński, Damian
Olsen, Hans
Osorio, M.
Paiva, Valeria de
Palmer, Doug
Paterson, Ross
Plandowski, Wojciech
Proebsting, Todd
Pöial, Jaanus
Roomeldi, Raul
Sabelfeld, Victor
Sastry, A. V. S.
Schultz, J. W.
Shepherd, John
Sondergaard, Harald
Stuckey, Peter
Sundararajan, R.
Swanson, Mark
Szczepanska-Waserztrum
Südholt, Mario
Thomasset, Francois
Tupailo, Sergei
Vain, Jüri
Vandecasteele, Henk
Virot, Bernard
Weemeeuw, P.
Zachary, Joe

Table of Contents

Executable Specifications for Language Implementation

Uwe Kastens

Fachbereich Mathematik/Informatik
University of Paderborn, D-4790 Paderborn, F.R. Germany

Abstract. Generating programs from specifications is an ambitious task that is solved at most for restricted application domains. General solutions which are practically satisfying as well are hard to achieve. Language implementation is a field, where tools and toolsets are available which process executable specifications and derive language implementing programs (compilers or interpreters) from them.
In this paper we will study specification principles that contribute to the success of program generation in this application domain. Examples are taken from the specification techniques of the Eli-System. The task of language implementation is decomposed into subtasks which have well understood and sufficiently general algorithmic solutions. Hence the instances of subtasks for a particular language can be specified. Certain language concepts like scope rules can be understood as a combination and variation of some basic rules. This situation allows specifications on a rather high level and reuse of precoined solutions. Domain specific expert know-how embedded in a toolset can further raise the specification level. The presentation of such specification principles in language implementation may raise discussion whether and how they can be applied to other areas as well.

1 Introduction

In the reference manual of Z [8] Spivey characterizes formal specifications as follows:

"Formal specifications use mathematical notation to describe in a precise way the properties which an implementation must have, without unduly constraining the way in which these properties are achieved. They describe *what* the system must do without saying *how* it is to be done."

The abstraction of the *what* from the *how* shall achieve specifications that have a small cognitive distance to the system requirements and a large distance to an implementation. Such specifications are *declarative* rather than *operational*.

Specifications have an important role in the software life-cycle: They are a reference point for both requirements analysis and implementation, and are a valuable means of promoting a common understanding among all those concerned with the system.[8] Specifications serve for proving an implementation against the requirements with respect to certain properties, e. g. invariants on the system states, I/O relation of a function, or mutual exclusion of critical operations in a parallel system.

The role of specifications in software development is further increased if an implementation is derived by refinement of the specification. Each refinement step introduces a design decision moving towards an implementation while keeping the specified properties intact. If we could get to an implementation without augmenting the specification of system properties by design decisions explicitly we had an *executable specification*. It could serve either for prototyping or for generating software products, depending on the software quality of the implementation.

Executable specifications especially for rapid prototyping is the goal of specification languages classified as Very High Level Language (VHLL). Krueger [7] discusses VHLL like SETL, PAISLey, and MODEL under the aspect of software reuse. The reuse effect is achieved by the specification language compiler or interpreter. It makes the implementation decisions without involving the author of the specification.

General purpose specification languages, as those mentioned so far, are based on elementary mathematical concepts: sets, functions, and sequences for modelling data, and predicate logic for modelling properties of operations. Systems that interpret or compile such specifications have to use generally applicable implementation strategies. So on the one hand all aspects of a system have to be specified and refined down to those elementary concepts. On the other hand the efficiency of the automatically achieved implementation is at best acceptable for prototyping.

This situation can be dramatically improved if the problem domain is restricted to a certain application area: A system of that domain can be described in terms of a dedicated specification language. An *application generator* translates such a specification into an implementation [1]. Krueger [7] characterizes domains appropriate for application generators, if "many similar software systems are written, one software system is modified or rewritten many times during its lifetime, or many prototypes of a system are necessary to converge to a usable product". Report generators for data bases are a typical area for application generators [3].

Narrowing the problem domain yields important advantages for specification design and execution:

- A specification may refer to concepts that are well-understood in the domain and hence need not be further refined.
- A domain specific model for problem decomposition can induce a modular structure of the implementation without being specified explicitly for each system.
- Domain specific implementation techniques can be applied automatically.

Hence systems are described on a high level and the specifications are executable.

In this paper we discuss strategies for executable specifications in the domain of language implementation. Translator generation can also be considered as an instance of the application generator principle, although this research is much older than the application generator idea. More than forty years of research and practice in compiler construction have resulted in common understanding of task decomposition and of subproblems, powerful formal methods for problem description, and in systematic implementation techniques. Tools are available that generate implementations from specifications, hence achieve their executability. The domain is very broad, ranging from compilers and source-to-source translators for programming languages to the implementation of dedicated specification languages, as used for application generators.

In the following sections we emphasize the discussion of specification strategies applied in this application domain to achieve executability. We use Eli [2] [4] as an example for a system which integrates many generating tools, precoined solutions, and domain specific knowledge. A major design goal of Eli is to achieve executable specifications that have a small cognitive distance to requirements of its problem area. We have learned many aspects of the specification strategies discussed here from the experience in developing and using the Eli system.

2 Domain Specific Decomposition

Decomposition of problems into subproblems is a natural method for analysis and design. Different aspects of a problem are separated and described on a suitable level of abstraction. Solutions can be found for smaller units using different techniques suitable for the particular subtask. Modular structure of the implementation and its interfaces can be derived from the decomposition structure.

If the problem space is restricted to a certain application domain specification and solution can be supported by a domain specific decomposition model that can be applied for any particular problem instance of that domain. Many years of experience in the language implementation domain led to a generally accepted model for decomposition of compilation tasks, as shown in Figure 1 taken from [2]. That model is not restricted to programming language compilers: In case of arbitrary language translation or interpretation the transformation phase usually yields the final result, the encoding phase is left out.

The existence of a suitable domain specific decomposition model simplifies the development of particular problem specifications: The description of the model leads to a structured way of reasoning about the problem - even if users are not experienced in language design and translation: It becomes obvious that, for example, the form of tokens of the input language has to be specified, or rules for name analysis must be chosen if the language has named objects. The model suggests that these properties of the problem refer to different subtasks and that they are related by the representation of name tokens.

A domain specific decomposition also allows one to specify different subtasks via dedicated formal models using suitable specification languages: E. g. the form of tokens is described by regular expressions like

```
        Ident:  $[a-zA-Z][a-zA-Zo-9]*
```