Berthold Hoffmann
Bernd Krieg-Brückner (Eds.)

Program Development by Specification and Transformation

The PROSPECTRA Methodology, Language Family, and System

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, FRG

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

Volume Editors

Berthold Hoffmann
Bernd Krieg-Brückner
FB 3 Mathematik und Informatik, Universität Bremen
Postfach 33 04 40, D-28334 Bremen, Germany

CR Subject Classification (1991):D.2.1, D.2.3, D.2.4, D.2.6, D.2.10, D.2.m

ISBN 3-540-56733-X Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-56733-X Springer-Verlag New York Berlin Heidelberg

Typesetting: Camera ready by author
45/3140-543210 - Printed on acid-free paper

Preface

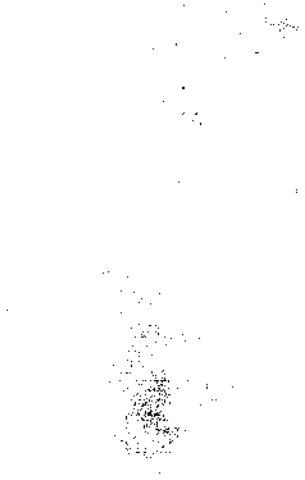
The PROSPECTRA project has been partially funded by the Commission of the European Communities under the ESPRIT Programme, ref. #390 and #835, from March 1985 to March 1990. Many people have contributed to the project (see *The PROSPECTRA Consortium* and *The PROSPECTRA Teams* on the next pages). The Consortium also very gratefully acknowledges the constructive contributions of the project officers from the Commission for PROSPECTRA, Dr. Pierre-Yves Cunin and Jack Metthey, and, last but not least, the Reviewers, Robert F. Maddock (IBM, Hursley), Professor Peter Pepper (Technische Universität Berlin), and Professor John Darlington (Imperial College, London), who have carefully, critically and benevolently guided the project through easy and hard times.

The objective of this documentation is a coherent presentation of the outcome of the project PROSPECTRA (PROgram development by SPECification and TRAnsformation) that aimed to provide a rigorous methodology for developing *correct* software and a comprehensive support system. The results are substantial: a theoretically well-founded *methodology* covering the whole development cycle, a very high-level specification and transformation *language family* allowing meta-program development and formalisation of the development process itself, and a prototype development *system* supporting structure editing, incremental static-semantic checking, interactive, context-sensitive transformation and verification, development of transformation (meta-) programs, version management, etc., with an initial library of some specifications and a sizeable collection of implemented transformations.

One intended audience for this documentation is clearly the academic community working in the areas of formal methods for software (and hardware) development, specification languages, theory of computation, semantics and verification, implementation of functional languages, structure editors, attribute grammars, advanced software engineering environments, etc. An even more important audience is the industrial community interested in the use of formal methods. It is still a long way to the widespread use of production-quality systems employing formal methods to increase correctness, reliability, and safety of systems, and productivity of developers. The PROSPECTRA Consortium has made a conscious effort of technology transfer, trying to implement the state-of-the-art, in a realistic setting. The prototype system, a "PROSPECTRA workstation", allows serious experimentation to enable feedback for extensions and improvements (that are undoubtedly needed). Eventually, we see various classes of PROSPECTRA users, with potentially distinct abilities and educational background: the PROSPECTRA system developers, the developers of transformations and development methods, the developers of (generically re-usable) specifications, and the software developers (end users). At the moment, the system is really only usable externally for benevolent experimenters due to its size and complexity of integration (coming from many development sites). We hope for a new version in the near future, however, based on the extensive experience with PROSPECTRA, as related work at Universität Bremen is presently funded by the Bundesministerium für Forschung und Technologie in the national project KORSO („Korrekte Software").

This volume contains three Parts. Part I contains a description of the PROSPECTRA Methodology of specification, transformation and verification, including the catalogue of presently available transformations. Part II contains a description of the PROSPECTRA Language Family: a rationale for the language subsets and their relationship, reference manuals for concrete syntax, informal semantics, abstract syntax and static semantic attributes, and a formal definition of the semantics of the specification subset. Part III contains a description of the PROSPECTRA System: a rationale for the uniform system structure, a short overall users´ guide, and reference manuals for the various system components.

Bremen, March 1993 — Bernd Krieg-Brückner, Berthold Hoffmann



The PROSPECTRA Consortium

Professor Bernd Krieg-Brückner
(Project Director)
FB3 Mathematik und Informatik
Universität Bremen
Postfach 330 440
D- 28334 Bremen

Universität Bremen *(Prime Contractor)*

Professor Harald Ganzinger
now at:
Max-Planck-Institut für Informatik
Im Stadtwald
D- 66123 Saarbrücken

Universität Dortmund

Professor Manfred Broy
now at: Institut für Informatik
Technische Universität München
Arcisstraße 21
D- 80290 München

Universität Passau

Professor Reinhard Wilhelm
FB14 - Informatik
Universität des Saarlandes
Postfach 1150
D- 66041 Saarbrücken

Universität des Saarlandes

Professor Andrew D. McGettrick
Computer Science Department
University of Strathclyde
Livingstone Tower, 26 Richmond St.
UK- Glasgow G1 1XH

University of Strathclyde

Dr. Emmanuel Fermaut
Syseca Logiciel
315 Bureaux de la Colline
F- 92213 St Cloud Cedex

Syseca Logiciel

Einar W. Karlsen
CASE Division
Computer Resources International A/S
(Dansk Datamatik Center)
Bregnerødvej 144
DK- 3460 Birkerød

Computer Resources International

Angel Perez Riesco
Research Center
Alcatel Standard Eléctrica, S. A.
Ramirez de Prado, 5
E- 28045 Madrid

Alcatel Standard Eléctrica SA

Professor Fernando Orejas
Departamento de Lenguajes
y Systemas Informáticas
Universitat Politécnica de Catalunya
Pau Gargallo 5
E- 08028 Barcelona

Universitat Politécnica de Catalunya
(Subcontractor)

The PROSPECTRA Teams

Prof. Bernd Krieg-Brückner, Dr. Berthold Hoffmann, Bernd Gersdorf,
Frank Drewes, Yulin Feng, Jörn von Holten, Stefan Kahrs, Wei Li, Junbo Liu,
Detlef Plump, Zhenyu Qian, Richard Seifert, Elisabeth Swart

Universität Bremen

Prof. Harald Ganzinger, Hubert Bertling,
Dr. Michael Hanus, Renate Schäfers

Universität Dortmund

Prof. Manfred Broy, Thomas Grünler, Dr. Friederike Nickl,
Michael Breu, Frank Dederichs, Rainer Weber

Universität Passau

Prof. Reinhard Wilhelm, Reinhold Heckmann, Dr. Ulrich Möncke,
Martin Alt, Andreas Fecht, Christian Ferdinand, Andreas Hense,
Peter Lipps, Stefan Pistorius, Georg Sander, Beatrix Weisgerber

Universität des Saarlandes

Prof. Andrew D. McGettrick, Owen Traynor,
Charles Chen, David Duffy, Joseph McLean

University of Strathclyde

Dr. Emmanuel Fermaut, Alain Marcuzzi,
Hérvé Bazin, Pierre Boulle, Ian Campbell, Dominique Girard, Dominique Houdier, Dr. Arnaury Legait,
Bernard Mathae, Elaine Morcos, Dr. Olivier Roubine, Jean-Luc Saouli, Chantal Vilhet

Syseca Logiciel

Dr. Georg Winterstein,
Peter Dencker, León Treff, Erich Zimmermann

Systeam KG Dr. Winterstein*

Einar W. Karlsen,
Jesper Andersen, Nicola Botta, Jesper Jørgensen
Steen Lynenskjøld, Claus Bendix Nielsen

Computer Resources International A/S

Angel Perez Riesco, Pedro de la Cruz Ramos,
Maria Dolores Hinojal, Alicia Lopez, Juan Antonio de Miguel, José Luis Mañas,
Carlos Muñoz, Miguel Muñoz†, Rafael Perez Gonzales, José Manuel del Prado, José Miguel Pinilla

Alcatel Standard Eléctrica SA

Prof. Fernando Orejas,
Marisa Navarro, Maria Pilar Nivela, Roberto Nieuvenhuis, Ricardo Peña

Universitat Politécnica de Catalunya

---

* past member team

Contents

PART II: LANGUAGE FAMILY

PART III: SYSTEM

Author Index

1. Introduction

Bernd Krieg-Brückner, Universität Bremen

This chapter gives a tutorial introduction to the Methodology. It serves as an overall rationale for the PROSPECTRA Project and relates this part to those on the Language Family and the System. In the methodology of PROgram development by SPECification and TRAnsformation, algebraic specifications are the basis for constructing *correct* and efficient programs by gradual transformation. The combination of algebraic specification and functionals increases abstraction, reduces development effort, and allows reasoning about correctness and direct optimisations. The uniformity of the approach to program and meta-program development is stressed (cf. also the chapter on Uniform Transformational Development in part III chapter 1).

1.1. Overview

The project PROSPECTRA ("PROgram development by SPECification and TRAnsformation") aims to provide a rigorous methodology for developing *correct* software and a comprehensive support system. From 1985 to 1990, it was sponsored by the Commission of the European Communities in the ESPRIT Programme, ref. #390 and #835, as a cooperative project between Universität Bremen (Prime Contractor), Universität Dortmund, Universität Passau, Universität des Saarlandes (all D), University of Strathclyde (GB), SYSECA Logiciel (F), Computer Resources International (DK), Alcatel Standard Eléctrica S.A. (E), and Universitat Politécnica de Catalunya (E) (cf. [Krieg-Brückner 88a, 89a, b, 90], [Krieg-Brückner 91b] (of which this combined volume is a revised edition), [Krieg-Brückner et al. 91], [Karlsen, Krieg-Brückner, Traynor 91], [Liu, Traynor, Krieg-Brückner 92], and the bibliography in part III chapter 7).

The Methodology of Program Development by Transformation (based on the CIP approach of TU München, see e.g. [Bauer 79], [Bauer et al. 85-89]) integrates program construction and verification during the development process. User and implementor start with a formal specification, the interface or "contract". This initial specification is then gradually transformed into an optimised machine-oriented executable program. The final version is obtained by stepwise application of transformation rules. These are applied by the system, with interactive guidance by the implementor, or automatically by compact transformation scripts. Transformations form the nucleus of an extendible knowledge base. Any kind of activity is conceptually and technically regarded as a transformation of a "program" in one of the system components. This provides for a uniform user interface, reduces system complexity, allows the construction of system components in a highly generative way, and is the basis for generalisation of specification, transformation, and command language, even library access, into a single framework.

Overall, PROSPECTRA has achieved a powerful specification *and* transformation *language* with well-defined semantics that reflects the state-of-the-art in algebraic specification combined with higher order functions. In addition, a comprehensive *methodology* covering the complete life-cycle (including re-development after revisions), integrating verification in a realistic way, supporting the development process as a computer-aided activity, and giving hope for a comprehensive formalisation of programming knowledge. A *prototype system* is operational, with a uniform user interface and library management including version and configuration control, that gives complete support and control of language and methodology to ensure correctness.

1.1.1. Overview of Part I

This book contains three Parts. Part I contains a description of the PROSPECTRA Methodology of specification, transformation and verification, including the catalogue of presently available transformations. Part II contains a description of the PROSPECTRA Language Family: a rationale for the language subsets and their relationship, reference manuals for concrete syntax, informal semantics, abstract syntax and static semantic attributes, and a formal definition of the semantics of the specification subset. Part III contains a description of the PROSPECTRA System: a rationale for the uniform system structure, a short overall users guide, and reference manuals for the various system components.

The intended audience for part I, on the PROSPECTRA Methodology, is the program developer.

Chapter 1 gives a tutorial introduction to the methodology, including a small representative example for program development by transformation. It serves as an overall rationale for the PROSPECTRA Project and relates this part I to those on the Language Family and the System.

Chapter 2 describes the specification approach used. It is written for a reader who is familiar with the general concepts of algebraic specification and wants to learn about the particular approach used in the PROSPECTRA Project and its extensions over more conventional approaches, such as loose specifications, partial, higher-order and non-strict functions, notably for the description of distributed systems.

Chapter 3 contains a Reference Manual of the Transformations that are presently available in the System, intended for the program developer. The individual transformations are described in a tutorial style using a semi-formal notation, with some examples. The catalogue is not complete but rather a collection of representative transformations. It will become more complete over time.

Chapter 4 is a tutorial introduction to the methodology of verification.

1.1.2. PROSPECTRA

Within chapter 1, the *objectives* of the PROSPECTRA methodology, its development model, algebraic specification and transformational program development are briefly summarised in section 1.2; the following subsections concentrate on particular extensions to classical algebraic specification and their relation to the methodology. An example illustrating the transformational approach, as supported by the PROSPECTRA system, is given in section 1.3.

1.1.3. Algebraic Specification and Functionals

Section 1.4 describes the combined advantages of functional programming and algebraic specification: a considerably higher degree of abstraction, avoiding much repetitive development effort, the use of homomorphic extension functionals as "program generators". The importance of the *combination* of algebraic specification with higher order functions should be stressed. The ability to specify *partial* higher-order functions (i.e. with conditions on functional parameters; see part II chapter 3) has been an important contribution of PROSPECTRA to the theory of algebraic specifications. The algebraic properties of functionals allow a high level of reasoning *about* functional programs, and permit general and powerful optimisations, supported by the PROSPECTRA approach.

1.1.4. Transformational Meta Program Development

The approach for meta-program development in PROSPECTRA, described in section 1.5, is to regard transformation rules as equations in an algebra of programs, to derive basic transformation operations

from these rules, to allow composition and functional abstraction, and to regard transformation *scripts* as (compositions of) such transformation operations. Using all the results from program development based on algebraic specifications and functionals we can then reason about the development of *meta-programs*, i.e. transformation programs or development scripts, in the same way as about programs. Homomorphic extension functionals are important for the concise definition of program development tactics.

Although section 1.5 focusses on meta-program development, it should be clear that the combined advantages of algebraic specification and higher order functions (described in section 1.4) apply to program and meta-program development in the same way. Similarly, all the transformation technology developed for program development can be carried over to meta-program development.

The meta-program development paradigm leads naturally to a *formalisation of the software development process* itself, described in section 1.6. A program development is a sequence of transformations. The system automatically generates a transcript of a development "history". A *development script* is a formal object that does not only represent a documentation of the past but is a plan for future developments. It can be used to abstract from a particular development to a class of similar developments, a *development method*, incorporating a certain strategy.

1.2. PROgram Development by SPECification and TRAnsformation

1.2.1. Objectives

Current software developments are characterised by ad-hoc techniques, chronic failure to meet deadlines because of inability to manage complexity, and unreliability of software products. The major objective of the PROSPECTRA project is to provide a technological basis for developing *correct* programs. This is achieved by a methodology that starts from a formal specification and integrates verification into the development process.

The initial *formal requirement specification* is the starting point of the methodology. It is sufficiently rigorous, on a solid formal basis, to allow verification of correctness during the complete development process thereafter. The methodology is deemed to be more realistic than the conventional style of *a posteriori* verification: the construction process and the verification process are broken down into managable steps; both are coordinated and integrated into an implementation process by *stepwise transformation* that guarantees *a priori* correctness with respect to the original specification. Programs need no further debugging; they are correct by construction. Testing is performed as early as possible by *validation* of the formal specification against the informal requirements (e.g. using a prototyping tool).

Complexity is managed by abstraction, modularisation and stepwise transformation. Efficiency considerations and machine-oriented implementation detail come in by conscious design decisions from the implementor when applying pre-conceived transformation rules. A long-term research aim is the incorporation of goal orientation into the development process. In particular, the crucial selection in large libraries of rules has to reflect the reasoning process in the development.

Engineering Discipline for Correct SW. The PROSPECTRA project aims at making software development an engineering discipline. In the development process, ad hoc techniques are replaced by the proposed uniform and coherent methodology, covering the complete development cycle. Programming knowledge and expertise are formalised as transformation rules and methods with the same rigour as engineering calculus and construction methods, on a solid theoretical basis.

Individual transformation *rules*, compact automated transformation *scripts* and advanced transformation *methods* are developed to form the kernel of an extendible knowledge base, the Method Bank, analogously to a handbook of physics. Transformation rules in the method bank are proved to be correct and thus allow a high degree of confidence. Since the methodology completely controls the system, reliability is significantly improved and higher quality can be expected.

Specification. Formal specification provides the foundation which enables the use of formal methods. High-level development of specifications and abstract implementations (a variation of "logic programming") is seen as the central "programming" activity in the future. In particular, the development of methods for "program synthesis", the derivation of constructive design specifications from non-constructive requirement specifications, is a present focus of research.

The abstract formal (algebraic) specification of requirements, interfaces and abstract designs (including concurrency) relieves the programmer from unnecessary detail at an early stage. Detail comes in by gradual optimising transformation, but only where necessary for efficiency reasons. Specifications are the basis for adaptations in evolving systems, with possible replay of the implementation from development histories that have been stored automatically.

The semantics of the specification language is based on the theory of algebraic specification (with looseness, partial functions, higher-order functions etc., see chapter 2), extended by constructs for predicative specification (pre- and post-conditions of functions).

A transformation is a development step producing a new program version by application of an individual transformation rule, or, more generally, a compact transformation "program" ("meta-" program, see sections 1.4, 1.5 below). Transformations preserve correctness and therefore maintain a tighter and more formalised relationship to prior versions. Their classical application is the construction of optimised implementations by transformation of an initial design that has been proved correct against the formal requirement specification. Further design activity then requires the selection of an appropriate rule, oriented by development goals, for example machine-oriented optimisation criteria.

Programming Language Spectrum: Ada and Anna. Targetting the general methodology and the support system to Ada [ADA 83] (with Anna as its complement for formal specification, see [Luckham et al. 87]) make it realistic for systems development. *PA*^{nn}*dA*, the PROSPECTRA (Anna/Ada subset) specification and programming language, covers the complete spectrum of language levels from formal specifications and applicative implementations to imperative and machine-dependent representations.

The target language has been Ada in the PROSPECTRA project, but the approach can be generalised to cover other targets as well (a translator to C is available). Stepwise transformations synthesise Ada programs such that many detailed language rules necessary to achieve reliability in direct Ada programming are obeyed *by construction* and need not concern the program developer. In this respect, the PROSPECTRA methodology makes a contribution to managing the complexity of Ada.

Research Consolidation and Technology Transfer. The PROSPECTRA project aims at contributing to the technology transfer from academia to industry by consolidating converging research in formal methods, specification and non-imperative "logic" programming, stepwise verification, formalised implementation techniques, transformation systems, and human interfaces.

Industry of Software Components. The portability of Ada allows pre-fabrication of software components. This is explicitly supported by the methodology. A component is catalogued on the basis of its interface.

Formal specification gives the semantics as required by and made visible to the user; the implementation is hidden and remains a (company) secret.

The methodology emphasises the *pre-fabrication* of generic, universally *(re-)usable* components that can be instantiated according to need. This will invariably cut down production costs by avoiding duplicate efforts. The production of perhaps small but universally marketable (Ada) components on a common technology base can also assist smaller companies in Europe.

Tool Environment. Emphasis on the development of a comprehensive support system is mandatory to make the methodology realistic. The system can be seen as an integrated set of advanced tools based on a minimal support environment, e.g. the ESPRIT Portable Common Tool Environment (PCTE). Because of the generative nature of system components, adaptation to future languages is comparatively easy.

The support of correct and efficient transformations is seen as a major advance in programming environment technology. The central concept of system activity is the application of transformations to trees. Generator components are employed to construct transformers for individual transformation rules and to incorporate the hierarchical approach of PA^{nn}dA (PROSPECTRA Anna/Ada), TrafoLa (the language of transformation descriptions), and ControLa (the command language); in fact, these turn out to be all sublanguages of the same language, for user program, transformation, proof, and system development. This integration and uniformity is seen as one of the major results of the PROSPECTRA project (cf. [Krieg-Brückner 88-92], [Karlsen, Krieg-Brückner, Traynor 91], [Krieg-Brückner et al. 91], [Liu, Traynor, Krieg-Brückner 92] and see part III chapter 1). Generators, in particular the Synthesizer Generator (cf. [Reps, Teitelbaum 88]), increase flexibility and avoid duplication of efforts; thus the overall system complexity is significantly reduced.

1.2.2. The Development Model

Consider a simple model of the major development activities in the life of a program:

Requirements Analysis

- Informal Problem Analysis
- Informal Requirement Specification

Development

- **Formal** Requirement Specification ⇑ *Validation*
- **Formal** Design Specification ⇑ *Verification*
- **Formal** Construction *by Transformation* ⇑ *Verification*

Evolution

- Changes in Requirements ⇒ **Re-Development** ⇑

The *informal requirements analysis* phase precedes the phases of the *development* proper, at the level of formal specifications and by transformation into and at the level(s) of a conventional programming language such as Ada. After the program has been installed at the client, no maintenance in the sense of conventional testing needs to be done; "testing" is perfomed *before* a program is constructed, at the very early stages, by validation of the formal requirement specification against the informal requirements.

The *evolution* of a program system over its lifetime, however, is likely to economically outweigh the original development by an order of magnitude. Changes in the informal requirements lead to re-development, starting with changes in the requirement specification. This requires re-design, possibly by *replay* of the original development (which has been archived by the system) and adaptation of previous designs or re-consideration of previously discarded design variants.

1.2.3. Specification

A requirement specification defines *what* a program should do, a design specification *how* it does it. The motivations and reasons for design decisions, the *why's*, are recorded along with the developments.

Requirement specifications are, in general, non-constructive: there may be no clue for an algorithmic solution of the problem or for a mapping of abstract to concrete (i.e. predefined) data types. It is essential that the requirement specification should not define more than the *necessary* properties of a program to leave room for design decisions. It is intentionally vague or *loose* in areas where the further specification of detail is irrelevant or impossible, i.e. it denotes a set of models (cf. [Krieg-Brückner 90]). In this sense, loose specification replaces non-determinacy, for example to specify an unreliable transmission medium in a concurrent, distributed situation [Broy 87d, 88, 89], [Dedcrichs 89].

From an economic point of view, overspecification may lead to substantial increase in development costs and inefficiency of execution of the program since easier solutions are not admissable. If the requirement specification is taken as the formal *contract* between client and software developer, then there should perhaps be a new profession of an independent *software notary* who negotiates the contract, advises the client on consequences by answering questions, checks for inconsistencies, resolves unintentional ambiguities, but guards against overspecification in the interest of both, client and developer. The answer of questions about properties of the formal requirement specification correspond to a *validation* of the informal requirement specification using a prototyping tool.

As an example take the specification of Booleans in (2-1), as it might appear for the standard Ada type. Some axioms (such as associativity, commutativity, distributivity) specify important properties of Booleans, but they are non-operational, whereas other equations can be interpreted as operational rewrite rules, see also (5-3) below.

Note that BOOLEAN is an algebraically specified Abstract Data Type (as the others below) such that its values can be manipulated in user-defined functions, etc., whereas axioms in the specifications are of a built-in type LOGICAL that denotes two-valued logic (without undefined). For better readability, the LOGICAL operators are written in the usual mathematical notation, e.g. ∨ instead of **or**. A symbolic style rather than a more conventional Ada oriented notation is used, e.g. —> instead of **return**, and, to exhibit the use of functionals, a notation with explicit Curry-ing to allow partial parameterisation.

In the example for natural numbers in (2-1), some general, non-operational properties are given first (more are needed to characterise the natural numbers completely). Then two alternative sets of rewrite rules are given (corresponding to the two boxes side by side; only a few rules are shown as an example): one for a linear presentation based on zero and succ as constructors, the other for a binary presentation based on zero, dble and dble1 as constructors. They play an important role for (automatic) simplification during transformation below. Each set represents a different design decision. Using the Conditional Equational Completion subsystem (cf. [Ganzinger 87]), each set of rewrite rules has been made terminating and confluent. The general properties are the basis for derivation of the more technical rewrite rules: cf. e.g. sqr x = x * x and the sets of corresponding rewrite rules for sqr.