Modular Compiler Verification

A Refinement-Algebraic Approach
Advocating Stepwise Abstraction
Müller-Olm, Markus:
Modular compiler verification: a refinement algebraic approach
advocating stepwise abstraction / Markus Müller-Olm. - Berlin;
Heidelberg ; New York ; Barcelona ; Budapest ; Hong Kong ;
London ; Milan ; Paris ; Santa Clara ; Singapore ; Tokyo : Springer,
1997
(Lecture notes in computer science ; 1283)
ISBN 3-540-63406-1

CR Subject Classification (1991): D.3.4, D.2.4, D.3.1, J.7, E3.1, C.3
ISSN 0302-9743

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 for prosecution under the German Copyright Law.

© Springer-Verlag Berlin Heidelberg 1997
Printed in Germany

Typesetting: Camera-ready by author
SPIN 10547711 06/3142 - 5 4 3 2 1 0 Printed on acid-free paper
After 40 years of practice and theory in compiler construction and 30 years of experience and teaching in software engineering we still observe that safety-critical high-level language programs are certified only together with the corresponding machine code. The reason is that certification institutions do not trust any compiler. And they are quite right: whereas errors detected in processor hardware are generally perceived as sensations, errors in software, even in system software, are commonplace.

It is high time to reverse this trend. Computer scientists should concentrate their abilities, experiences, and insights on the safe mastery of realistic system software. This particularly concerns realistic compilers for realistic programming languages running on hardware processors, as correct compilers play a central role in the construction of trustworthy application and system programs. Both application programmers and system software engineers need trusted development environments that permit them to concentrate on software specification and high-level implementation instead of wasting their time again and again with compilation problems and machine code inspection.

In order to construct fully correct and realistic compilers useful for safety-critical applications one must master two problems: compiling specification verification and compiler implementation verification. The first problem is to specify and prove semantically correct a mathematical translation function from high-level source programs to realistic target machine code. The second problem is to correctly refine and implement such a function either in a host language for which there is a running and trusted host compiler or in the machine language of a real host processor. Clearly, initially only the second way is trustworthy. M. Müller-Olm lays foundations for both problems.

One of the most characteristic aspects of M. Müller-Olm’s way of thinking is his special way of looking at the behavior of hardware processors. He derives successively more elegant views, which allow real processors to be treated as idealized machines with ideal properties. This enables compiling verification techniques to be applied that rely on such machines. Future further propagation of the hierarchy of views promises clearer insights and proofs for the translation of even more ambitious realistic programming languages.

Kiel, May 1997

Hans Langmaack
Preface

Whenever software is developed for safety-critical systems, trusted, verified compilers would be of great value because they would make it possible to reason about software correctness solely on the source language level. Many people, however, are rather skeptical when confronted with the idea of completely verified realistic compilers. They fear that proofs addressing the complexity of compilers generating actual machine code would become long and involved and, therefore, would hardly be convincing. Indeed, the available literature on compiler verification mostly concentrates on particular translation aspects in isolation, illustrated by toy source and target languages. More ambitious efforts usually stop at the level of some abstract machine.

This monograph aims to show that it is possible to reason convincing about correctness of translations to machine code of actual processors, and, even better, that this can be done in an illuminating and appealing manner. As a case study it provides the verified design of a code generator translating a real-time programming language to the Inmos Transputer. The success of such an effort crucially depends on appropriate structuring and the use of an adequate notation. Moreover, many theoretical topics are involved that are usually studied in isolation. Therefore, part of this monograph is devoted to consistently combining a number of subjects: algebraic reasoning with program-like combinators, data refinement theory, predicate transformer semantics, modeling of timing, and communication behavior. I hope that the resulting formal framework as well as the proof-engineering ideas incorporated in the verified code generator design are of interest even outside the realm of compiler verification.

This monograph is a revised version of my doctoral dissertation, the German equivalent of a Ph.D. thesis, which was submitted to the Technical Faculty of the Christian-Albrechts-University at Kiel and accepted in June 1996. I would like to thank my supervisor Hans Langmaack for support in many ways. I am also indebted to Martin Fränzle, who shared my office in the last few years and who was always willing to discuss technical and non-technical questions. He also gave a number of hints on a draft version that helped to improve the presentation.

The Provably Correct Systems (ProCoS) project provided the environment for writing my dissertation and I thank my colleagues for inspiration,
criticism, and discussion, in addition to the aforementioned persons in particular Tony Hoare, He Jifeng, Burghard von Karger, Ernst-Rüdiger Olderog, Anders P. Ravn, and Michael Schenke. I greatly acknowledge the financial support of the European Union which made this project possible by funding it in the ESPRIT programme (BRA 3104 and BRA 7071).

I am also grateful to the referees of my thesis, Hans Langmaack, Juraj Hromkovič, and Bernhard Steffen, for their time and enthusiasm and to the other members of the examination board, Rudolf Berghammer, Helmut Föll, Klaus Potthoff, Roland Schmidt, and Wolfgang Thomas. Last but not least I would like to thank Claudia Herbers for constant encouragement.

I hope that this monograph is a step towards changing people’s view of compiler verification: It should be regarded as an aid in correct design rather than as another burden for the compiler builder.

Passau, January 1997

Markus Müller-Olm
# Table of Contents

1. **Introduction** ......................................................... 1  
   1.1 Organization of This Book ........................................ 6  
   1.2 Related Literature on Code Generator Verification ............ 7  

2. **Complete Boolean Lattices** .................................... 9  
   2.1 Basic Definitions ................................................ 9  
   2.2 Functions ......................................................... 11  
      2.2.1 Functions from a Set to an Ordered Set or Lattice ....... 11  
      2.2.2 Functions Between Lattices ................................ 12  
   2.3 Fixpoints ......................................................... 14  

3. **Galois Connections** ............................................. 15  
   3.1 Definition and Basic Facts ...................................... 15  
   3.2 Distributive Galois Connections ................................ 17  
   3.3 Galois Connections Between Power Sets ....................... 19  
   3.4 Composing Galois Connections .................................. 20  

4. **States, Valuation Functions and Predicates** ................. 23  
   4.1 Variables ........................................................ 26  
   4.2 States ............................................................. 27  
      4.2.1 Variation .................................................... 27  
   4.3 Valuation Functions ............................................. 28  
      4.3.1 Projection and Lifting ..................................... 28  
      4.3.2 Substitution ............................................... 30  
      4.3.3 Independence ............................................... 32  
      4.3.4 Extending and Restricting the Alphabet ................... 34  
   4.4 Predicates ........................................................ 35  
      4.4.1 Quantification .............................................. 35  

5. **The Algebra of Commands** ...................................... 39  
   5.1 Definitions ...................................................... 41  
   5.2 Dualization ....................................................... 43  
   5.3 Simulations ....................................................... 44  
   5.4 Basic Homogeneous Commands .................................... 45
# Table of Contents

5.4.1 Multiple Assignments ........................................ 46
5.4.2 Assertions and Assumptions ................................. 48
5.4.3 Derived Laws for Basic Homogeneous Commands ......... 53
5.5 Basic Heterogeneous Commands ............................... 57
5.6 Distribution Properties ....................................... 67
5.6.1 Basic Commands ........................................... 67
5.6.2 Preservation of Distribution Properties ................. 67
5.6.3 Distribution Properties of Composition Operators ...... 68
5.7 Derived Commands ............................................. 69
5.7.1 Choice Assignments ...................................... 70
5.7.2 Conditionals .............................................. 78
5.7.3 Arrays ..................................................... 80
5.8 Discussion .................................................... 81

6. **Communication and Time** ................................... 85
6.1 An Operational Framework ................................... 87
6.1.1 Communication Commands ................................. 89
6.1.2 Loops ...................................................... 91
6.1.3 Remarks ................................................... 94
6.2 Definitions .................................................. 95
6.3 Input and Output Processes ................................ 96
6.4 Stop ......................................................... 98
6.5 Delays ....................................................... 98
6.6 Time Bounds ................................................ 101
6.7 Discussion ................................................... 103

7. **Data Refinement** ........................................... 105

8. **Transputer Base Model** ................................... 113
8.1 Basic Data-Types and Constants ............................ 114
8.2 Modeling the Transputer’s Components .................... 116
8.2.1 Registers ............................................... 116
8.2.2 Memory .................................................. 116
8.2.3 Links ..................................................... 118
8.3 Instructions and Operations ................................ 118
8.4 Capturing the Dynamic Behavior ............................ 119
8.5 Discussion ................................................... 124

9. **A Small Hard Real-Time Programming Language** ........ 127
9.1 Abstract Syntax ............................................. 129
9.2 Static Semantics ............................................. 130
9.3 Dynamic Semantics .......................................... 133
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.1</td>
<td>Symbolic Representation of Control Point</td>
<td>137</td>
</tr>
<tr>
<td>10.2</td>
<td>Large Operands</td>
<td>147</td>
</tr>
<tr>
<td>10.2.1</td>
<td>Some Properties of $prefix$</td>
<td>148</td>
</tr>
<tr>
<td>10.2.2</td>
<td>Performing the Abstraction</td>
<td>151</td>
</tr>
<tr>
<td>10.2.3</td>
<td>Operations</td>
<td>154</td>
</tr>
<tr>
<td>10.3</td>
<td>Workspace</td>
<td>156</td>
</tr>
<tr>
<td>10.4</td>
<td>Symbolic Addressing</td>
<td>160</td>
</tr>
<tr>
<td>10.4.1</td>
<td>Instructions and Operations</td>
<td>163</td>
</tr>
<tr>
<td>10.4.2</td>
<td>Extending and Restricting Dictionaries</td>
<td>169</td>
</tr>
<tr>
<td>10.5</td>
<td>Forgetting About Registers</td>
<td>175</td>
</tr>
<tr>
<td>10.6</td>
<td>Discussion</td>
<td>177</td>
</tr>
<tr>
<td>11.1</td>
<td>Compiling-Correctness Relations</td>
<td>179</td>
</tr>
<tr>
<td>11.2</td>
<td>Programs</td>
<td>181</td>
</tr>
<tr>
<td>11.3</td>
<td>Statements</td>
<td>183</td>
</tr>
<tr>
<td>11.4</td>
<td>Expressions</td>
<td>185</td>
</tr>
<tr>
<td>11.5</td>
<td>Monadic and Binary Operators</td>
<td>186</td>
</tr>
<tr>
<td>12.1</td>
<td>Translation Theorems</td>
<td>189</td>
</tr>
<tr>
<td>12.2</td>
<td>Preliminaries</td>
<td>189</td>
</tr>
<tr>
<td>12.3</td>
<td>Programs</td>
<td>193</td>
</tr>
<tr>
<td>12.4</td>
<td>Statements</td>
<td>194</td>
</tr>
<tr>
<td>12.5</td>
<td>Expressions</td>
<td>204</td>
</tr>
<tr>
<td>12.6</td>
<td>Monadic Operators</td>
<td>207</td>
</tr>
<tr>
<td>12.6.1</td>
<td>Binary Operators</td>
<td>209</td>
</tr>
<tr>
<td>12.6.2</td>
<td>Arithmetic Operators</td>
<td>209</td>
</tr>
<tr>
<td>12.6.3</td>
<td>Relational Operators</td>
<td>212</td>
</tr>
<tr>
<td>13.1</td>
<td>Boolean Operators</td>
<td>213</td>
</tr>
<tr>
<td>13.1</td>
<td>A Functional Implementation</td>
<td>215</td>
</tr>
<tr>
<td>13.2</td>
<td>Syntactic Dictionaries</td>
<td>216</td>
</tr>
<tr>
<td>13.3</td>
<td>Refined Correctness Relations</td>
<td>217</td>
</tr>
<tr>
<td>13.4</td>
<td>Refined Translation Theorems</td>
<td>218</td>
</tr>
<tr>
<td>13.4.1</td>
<td>Compiling Functions</td>
<td>220</td>
</tr>
<tr>
<td>13.4.2</td>
<td>Constructive Translation of Conditionals</td>
<td>225</td>
</tr>
<tr>
<td>13.5</td>
<td>Constructive Translation of Loops</td>
<td>226</td>
</tr>
<tr>
<td>13.5</td>
<td>Discussion</td>
<td>230</td>
</tr>
<tr>
<td>14.1</td>
<td>Conclusion</td>
<td>233</td>
</tr>
<tr>
<td>14.1.1</td>
<td>Topics for Future Research</td>
<td>236</td>
</tr>
<tr>
<td>14.1.2</td>
<td>Extensions</td>
<td>236</td>
</tr>
<tr>
<td>14.1.3</td>
<td>Improvements</td>
<td>237</td>
</tr>
<tr>
<td>14.1.3</td>
<td>Modifications</td>
<td>237</td>
</tr>
<tr>
<td>References</td>
<td>239</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>-----</td>
<td></td>
</tr>
<tr>
<td>Index</td>
<td>245</td>
<td></td>
</tr>
</tbody>
</table>
List of Figures

7.1 Proof obligation for operations ........................................... 106
10.1 Processes constituting the abstraction levels ...................... 136
10.2 Illustration of $A(a, b)$ .................................................... 138
List of Tables

5.1 Distribution properties of basic commands .......................... 67
5.2 Preservation of distribution properties .............................. 68
5.3 Distribution properties of predicate transformer operators ...... 70

8.1 Axioms for various direct functions ................................. 122
8.2 Axioms for various operations ........................................ 123

9.1 Abstract syntax of TPL .................................................. 130
9.2 Static semantics of TPL .................................................. 132
9.3 Dynamic semantics of TPL .............................................. 133

10.1 Illustration of abstraction levels ...................................... 137
10.2 Approximations for instructions I .................................... 144
10.3 Approximations for operations I ..................................... 146
10.4 Approximations for instructions II ................................... 153
10.5 Approximations for operations II .................................... 156
10.6 Approximations for instructions III .................................. 159
10.7 Approximations for operations III .................................. 160
10.8 Approximations for instructions IV .................................. 164
10.9 Approximations for operations IV ................................... 166

13.1 Signature of compiling functions .................................... 221
13.2 Compiling clauses for programs and statements except loops ... 222
13.3 Compiling clause for loops ............................................ 223
13.4 Compiling clauses for expressions ................................... 223
13.5 Clauses for initialization code and operators ...................... 224
1. Introduction

Much research has been performed in the last decades on verified design of programs. But to establish properties rigorously for high-level programs is worth the effort only if we can ensure that these properties transfer to the machine code that is finally executed. The latter usually is generated from the high-level program by a compiler. As inspection of the generated machine code is tedious and error-prone, it should be avoided by use of trusted compilers. In the area of safety-critical systems, certification could be performed on the source code level if trusted compilers were available, which promises to be less time-consuming and thus less costly as the current practice.

A common approach to increase confidence into a compiler is to compile a collection of test programs, to execute the resulting object code, and to inspect the results. It is questionable whether such a validation procedure can replace target code inspection in safety-critical software development, as test programs usually exhibit rather simple behavior and will not necessarily catch intricate timing and synchronization errors. Hence, development of a reliable compiler for a real-time programming language in our opinion should include formal verification of its vital constituents, in particular of its code generator.

Although the idea of mathematically verified compilers dates back at least to the sixties [54], the complete verification of realistic compilers is still a challenge. Most documented work on compiler verification heads for mathematical understanding of typical or semantically intricate implementation mechanisms. Therefore, generally source and target language are chosen in order to illustrate just the aspects under consideration. In particular, target code for commercially available processors rarely is formally investigated. One can argue that the application to actual machine code is straightforward but burdens the verification with many details of little interest, an attitude which is quite reasonable from a mathematician's point of view.

Besides mathematical insight, however, there is the more practical reason mentioned above for an interest in compiler verification: to justify the correctness of compiler-generated machine code from the correctness of the source code. Such a practical application of compiler verification calls for investigation of realistic processors and actual machine code.
1. Introduction

This book provides a case study for the verified design of a code generator translating to an actual processor. The source language is essentially the language of while programs extended by communication statements and upper bound timing, and the target architecture is the Inmos Transputer [46]. The effort put into such a verification is, of course, not so much justified by the particular correctness claim that is proved but by the methodological insights that support and simplify a later undertaking of a similar kind.

The work has been performed in the context of the ProCoS project [13], a basic research action that has been funded by the European Union in the ESPRIT programme from 1989 until 1995 in two phases. ProCoS is an acronym for 'Provably Correct Systems' and the project partners in the second phase have been research groups from the universities in Kiel (Germany, site leader: H. Langmaack), Lyngby (Denmark, site leader: A.P. Ravn), Oldenburg (Germany, site leader: E.-R. Olderog), and Oxford (England, site leader: C.A.R. Hoare). The objective of ProCoS was 'to advance the state of the art of systematic design of complex heterogeneous systems, including both software and hardware; in particular to reduce the risk of error in the specification, design and implementation of embedded safety-critical systems' (cited from ProCoS's technical annex). More specifically, its goal was to develop a tower of notations with a consistent mathematical interpretation that can advantageously be used to document the development process from requirements capture down to implementation by machine code or task-specific hardware [35]. Compiler verification has been an important subtask, for which the Kiel group had the main responsibility.

A difficulty when verifying code for real processors is the large amount of machine-specific details that must be mastered: addressing modes, side effects of instructions, allocation of memory and registers, representation of data, restricted address space, no a-priori separation of program and data store, etc. It is essential to structure the entire proof appropriately such that small parts can independently be approached and checked.

The global structure of compilers is classical; they consist of a scanner, a parser, a well-formedness checker, and a code generator. Like most other work on compiler verification, we concentrate on the correctness proof for code generators, particularly on the semantic correctness of the generated code, because construction of scanners and parsers has been studied in the literature extensively.

Our main emphasis is on structuring code generator verification appropriately when translating to actually available hardware. The chosen structure must support the reader in checking the considerations in detail in order to allow a trustworthy certification of the correctness claim. Therefore, the verification must not consist of a few large monolithic proofs but must be split into (a larger number of) small proofs that can independently be examined. One might object that human control of a verification intended to validate the detailed correctness of a compiler is inherently unreliable due to its size
and that the use of mechanical provers is a must. But this is not at all an
argument against structuring: experience suggests that a good structure often
is the key to the success of a mechanical proof attempt.

But a structured approach to verification should also support the making
of the proof. One of the merits of a correctness proof is that assumptions
that are left implicit in informal reasoning must be made explicit. Thus after
the proof has been completed the side conditions required for execution of
the code and the invariants kept by the code are precisely documented. The
necessity of explication, however, is one of the sources for the complexity
of such a proof. It is not easy to guess a precise formulation of the side
conditions and the invariants together with correct code consistently, before
starting the proof. Therefore, it is advisable to design the code generation
map simultaneously together with the correctness property and the proof, in
order to avoid iteration of proof attempts. We are heading for a structure of
presentation and preparation that fosters a separation of concerns, such that
the writer as well as the reader can concentrate on one particular aspect of
the translation at a given time. Ultimate goal of the research reported here
is a methodology for the correct construction of code generators.

Of course the construction of a verified code generator does not start from
scratch but from a rough intuitive understanding gained from prior compiler
building experience and tradition. A useful methodology for the correct con-
struction of code generators must allow to make precise these informal ideas
in a stepwise and incremental fashion. Let us sketch the proposal that is exem-
plified in this book. The core is the stepwise derivation of increasingly abstract
views to the target processor's behavior from a base model of its execution
cycle. Each abstraction step allows to treat one particular aspect of transla-
tion or machine program execution in isolation. Then compiling-correctness
relations are defined that specify the intended semantic relationship between
source and target code, which is largely simplified by the availability of the
abstract views. Afterwards, concrete code patterns are studied by means of
theorems about the compiling-correctness relations. From these translation
theorems a code generator can be implemented without further semantic con-
sideration.¹

It is natural to think of instructions of von Neumann machines as assign-
ments to machine components like accumulators and store. Hence, the
effect of machine instructions can conveniently be described by imperative
programs. E.g. the effect of the Transputer instruction \texttt{1dc(1)}, which loads
the constant value 1 to the accumulator called A, and moves A's contents to

¹ In order to construct a verified code generator, the implementation must, in
principle, be done in machine code (for a trusted processor) because – at least
for the first program of this kind – we cannot rely on a trusted compiler. The
Verifix project [28] proposes to reduce the amount of manual work necessary for
construction of a first, verified implementation in machine code by a bootstrap
approach in multiple stages.
accumulator B, as well as B’s contents to accumulator C, can be represented by the multiple assignment

\[ E(\text{lda}(1)) = A, B, C := 1, A, B. \]

Similarly, the effect of st1(x), writing A’s contents to variable x, moving B’s value to A, C’s value to B, and an unspecified value to C, can be described by

\[ E(\text{st1}(x)) = x, A, B := A, B, C ; C := ? , \]

where C := ? abbreviates the nondeterministic choice between all possible assignments to C. If semantics of machine instructions is captured by imperative program fragments, refinement algebra of imperative programs [42] can be used to prove that certain machine instruction sequences refine, i.e. implement, certain source programs. The following calculation, for instance, shows that the code sequence \( (\text{lda}(1), \text{st1}(x)) \), assumed to have the same meaning as the sequential composition of the effects \( E(\text{lda}(1)) \) and \( E(\text{st1}(x)) \), is correct target code for the assignment \( x := 1 \):

\[
E(\text{lda}(1)) ; E(\text{st1}(x))
\]

\[
= \text{[Identities above]}
\]

\[
A, B, C := 1, A, B ; x, A, B := A, B, C ; C := ?
\]

\[
= \text{[(Combine-assign), (Identity-assign)]}
\]

\[
x, A, B, C := 1, A, B, C ; C := ?
\]

\[
= \text{[(Cancel-assign), (Identity-assign)]}
\]

\[
x := 1 ; C := ?.
\]

For the moment the additional effect on the accumulator C is taken to be irrelevant. In this proof we have used the assignment laws

\[
(\text{Identity-assign}) \quad (x := e) = (x, y := e, y)
\]

\[
(\text{Combine-assign}) \quad (x := e ; x := f) = (x := f[e/x])
\]

\[
(\text{Cancel-assign}) \quad (x, y := e, f ; y := ?) = (x := e ; y := ?)
\]

where \( f[e/x] \) denotes substitution of \( e \) for \( x \) in expression \( f \) (\( x \) and \( y \) stand for disjoint lists of variable and \( e \) and \( f \) for lists of expressions of corresponding type).

The above little calculation illustrates a basic idea of our approach, viz. to use an imperative meta-language and refinement laws as proposed by C.A.R. Hoare in [41]. But, of course, the presentation up to now is oversimplified. Firstly, the model of the instruction’s effect is too abstract. For example, the Transputer instructions refer to memory locations basically, not to variable identifiers as we assumed in the description of st1(x), and a machine program basically is not a separate entity but the executed instructions are taken from memory, thus running a risk of being overwritten. Secondly, a number of unformalized assumptions have been made in the surrounding text, e.g. that concatenation of machine programs corresponds to sequential composition.
Clearly, an abstract model of the target processor simplifies the compiler verification. But if considerations are based on such a model alone, there is a severe danger of unsoundness because the postulated model might fail to provide a safe abstraction of the processor's actual behavior. To avoid this danger in our case study, we interface directly to the Transputer's documentation and start from a semi-formal model given by Inmos, the manufacturer of the Transputer, in [46]. However, a direct application of this model in a compiler proof results in very long and tedious calculations, which would seriously affect credibility of the proofs. How can we combine simplicity and conciseness of proofs with realistic modeling of the processor?

As mentioned, the idea is to derive a hierarchy of mutually consistent, increasingly abstract views to the Transputer's behavior, starting from bit-code level up to assembly levels with symbolic addressing. Afterwards we can choose for each proof task the model that allows the simplest proof or even mix reasoning at different abstraction levels without risking inconsistencies or unsoundness.

In the tradition of the refinement calculus we use a notation similar to an imperative programming language furnished with a refinement relation [5, 6, 61, 65] as meta-language for describing the various Transputer models and the abstraction maps. The various theorems on abstractions and code sequences are proved by short calculations using algebraic laws about the imperative constructs as proposed in [42, 43]. The imperative notation is interpreted by predicate transformers as in E.W. Dijkstra's wp-calculus. This is classical for internal constructs like assignments [6, 20, 21, 61, 65]. But we extend this to communication and timing by using two distinguished variables $hst$ (for modeling the communication history) and $clk$ (for modeling the runtime consumption) in a specific way. Non-$\perp$-strict predicate transformers are exploited to assign a reasonable total correctness semantics to non-terminating commands. Following the example of R.J.R. Back and J. von Wright [5], we perform the reasoning in the heterogeneous algebra of monotonic predicate transformers over different state spaces. For performing the abstractions we use a variant of the data refinement theory of R.J.R. Back [5], P.H.B. Gardiner & C.C. Morgan [25], and J.M. Morris [66].

The translation task considered in this book is modest w.r.t. the level of the source language; the complexity mainly stems from the investigation of actual machine code. But there are two aspects of broader interest. On the one hand the source language allows to state upper bound requirements for the execution time of basic blocks, the validity of which must be checked by the compiler. This is complicated by an idealization offered in the source language that is essential for the convenient programming of real-time programs in our opinion, viz. that internal computation is immediate and time proceeds only in communication statements. We can offer this idealization because only the communication behavior of programs is externally observable. The compiler
has to distribute the actual execution time of the code implementing internal activity to subsequent communications.

The other interesting translation aspect is that we do not assume that a generally acceptable failure behavior is available. Classically, stopping of programs is assumed to be reasonable in failure situations like stack overflow or arithmetic under- or overflow. This assumption, however, often is wrong in the context of reactive programs: certainly a control program in an aeroplane must not stop under any circumstances. Consequently, machine restrictions must either be reflected on the source language level or checked by the compiler. The former approach allows the program designer to prove that the machine restriction does not affect correctness of the program, which is appropriate e.g. for the restricted arithmetic available on a specific processor. The latter is a reasonable strategy for the storage consumption of programs but excludes e.g. unrestricted use of recursion in the source language.

1.1 Organization of This Book

Almost half of this book is devoted to providing a firm foundation for calculations with the chosen imperative notation. In Chap. 2 we summarize basic results about complete Boolean lattices. The fundamental notion of Galois connections is discussed in Chap. 3. Chapter 4 defines and studies states, valuation functions and predicates. In particular, various ‘typed’ quantification operators are introduced. In Chap. 5 the basic notations of the imperative meta-language are defined and numerous laws are established. In Chap. 6 we extend this basic language with communication and timing primitives. In particular, we present a trace semantics for communicating processes in the space of predicate transformers which reasonably models non-terminating processes that communicate infinitely often. For this behalf the classic notion of total correctness is extended to communicating processes, which is motivated and studied in an operational framework. Chapter 7 is concerned with data refinement. Essentially, we reformulate the definition of J.M. Morris [66] and P.H.B. Gardiner & C.C. Morgan [25, 63] on the process level. This concludes the first part that provides the notational and mathematical basis for the second part that is concerned more directly with code generator correctness.

In Chap. 8 we formulate the most concrete model of the target processor, viz. the Inmos Transputer. It is obtained by re-stating information from the Transputer instruction set manual [46] as refinement axioms about a certain collection of processes. Chapter 9 defines the source language TPL, a prototypic hard-real time programming language. In Chap. 10 we construct in successive steps more abstract views to the behavior of the Transputer that are exploited when defining in Chap. 11 the semantic compiling-correctness relations that must hold between source and target code. By means of theorems about these relations, concrete correct code patterns are studied in
Chap. 12. How a code generator can be implemented from this collection of translation theorems is illustrated in Chap. 13 by the construction of a functional program. We finish with concluding remarks in Chap. 14. An index at the end of this book is particularly intended to support finding definitions and laws.

1.2 Related Literature on Code Generator Verification

The idea to specify a machine by a high level program is old and present already in the concept of micro-programming [86]. G.M. Brown, A.J. Martin and others [15, 36, 52] use such descriptions as starting point for hardware design and C.A.R. Hoare, H. Jifeng and A. Sampaio [41, 43, 79] propose to utilize them together with a refinement algebra related to the source language [42, 60] for reasoning about code generator correctness. We adopt this proposal but apply it in a different way. From more classical work about compiler correctness [48, 54, 64, 57, 71, 78, 84] we are particularly distinguished by aiming at code for an actually commercially available processor and not for idealized hardware, which puts an emphasis on modularity and on readability of the meta-language used. Another difference to classical methods is that we use refinement as the correctness notion instead of semantic equivalence, which is also borrowed from [43]. This allows a proper treatment of under-specification in the source language’s semantics (e.g. of uninitialized variables) and accommodates modularization. Like the work at CLI (Computational Logic Inc.) [7, 59] on the ‘verified stack’ we put emphasis on consistent interfaces to higher and in particular lower levels of abstraction.

E. Börger, I. Duradonović and D. Rosenzweig [11, 12] are concerned with proving correct compilation of Occam to the Transputer. Their emphasis, however, is more on a mathematical understanding of its specific multitasking and communication mechanisms, while we are more interested in the common phenomena arising when translating to actually available von Neumann processors. The recent impressive work on VLisp by J.D. Guttman, D.P. Oliva, J.D. Ramsdell, V. Swarup and M. Wand [33, 34, 73] is concerned with a verified translator for Scheme, a functional Lisp-like language. The final abstract machine is rather close to actual hardware given the abstractness of the source language Scheme but is still more abstract than code for commercially available processors. They propose to use an operational style of reasoning for the verification at lower levels while we advocate a more abstract denotational style even there. Some of their abstractions rely on the existence of an acceptable failure behavior, which prohibits an immediate application of their work in the area of embedded safety-critical control programs.
2. Complete Boolean Lattices

We assume familiarity with the basic theory of complete Boolean lattices. To make this book reasonably self-contained, however, we repeat the basic definitions and facts in this section. An in-depth exposition of lattice theory is provided by the classic books of G. Birkhoff [9] and G. Grätzer [29]. We also partly follow the introductory sections of [6]. All the results are standard and are therefore reported here without proof.

2.1 Basic Definitions

This section contains the basic definitions and facts of lattice theory. We cite them from the first chapter of [29] mainly.

A partial order on a non-empty set \( A \) is a binary relation \( \leq \) on \( A \) that is reflexive, antisymmetric and transitive, i.e. that satisfies the following three axioms.

- **Reflexivity:** for all \( a \in A \): \( a \leq a \).
- **Antisymmetry:** for all \( a, b \in A \): \( a \leq b \land b \leq a \Rightarrow a = b \).
- **Transitivity:** for all \( a, b, c \in A \): \( a \leq b \land b \leq c \Rightarrow a \leq c \).

If \( \leq \) is a partial order on a set \( A \) then we call the pair \( (A, \leq) \) a partially ordered set. If \( \leq \) is clear from context, we often call \( A \) a partially ordered set omitting an explicit reference to \( \leq \).

Assume \( (A, \leq) \) is a partially ordered set and let \( B \subseteq A \) and \( a \in A \). Then \( a \) is called an upper bound of \( B \) if \( b \leq a \) for all \( b \in B \). An upper bound \( a \) of \( B \) is called the least upper bound or join of \( B \) if it is smaller than any other upper bound. By antisymmetry of \( \leq \), the least upper bound is unique if it exists. We denote the least upper bound of \( B \) (if it exists) by \( \bigvee B \). Dually, \( a \) is called a lower bound of \( B \) if \( a \leq b \) for all \( b \in B \), and called the greatest lower bound or meet of \( B \) if it is a lower bound greater than any other lower bound. Like the least upper bound the greatest lower bound of \( B \) is unique if it exists and is then denoted by \( \bigwedge B \). \( B \) is called a chain if it is non-empty and all its elements are comparable w.r.t. \( \leq \), i.e. if for all \( b, c \in B \), \( b \leq c \) or \( c \leq b \).

A partially ordered set \( (L, \leq) \) is called a lattice if for any two elements \( a, b \in L \) the least upper bound \( \bigvee\{a, b\} \) as well as the greatest lower bound