Provably Correct Systems
ProCoS II
ESPRIT Basic Research 7071

Automated Code Generator Verification Based on Algebraic Laws

Karl-Heinz Buth

Date: September, 1995

Doc. Id.: [Kiel KHB 5/1]

Doc. type:	Report
Status:	Final
Activity-alloc:	Verification support
Distribution:	Free
Copyright © 1995	Christian-Albrechts-Universität Kiel

Document History

This is the first edition.

Document Purpose

This report presents ways to provide automated support for code generator verification as described by Markus Müller-Olm in [MO95] using the Prototype Verification System (PVS, see [Rus93]).

Project Sponsors

This report reflects work which is partially funded by the Commission of the European Communities (CEC) under the ESPRIT programme in the field of Basic Research Project No. 7071: "**ProCoS II**: Provably Correct Systems". The author is supported by the Deutsche Forschungsgemeinschaft (DFG), grant La 426/12-2.

Site Address

Christian-Albrechts Universität zu Kiel
Institut für Informatik und Praktische Mathematik
Preußerstraße 1–9
D-24105 Kiel
Germany
Tel: +49-431-5604-28
Fax: +49-431-5661-43
E-mail: `procos@informatik.uni-kiel.d400.de`

Partners

Oxford University – coordinator (OU)
Technical University of Denmark (ID/DTH)
Christian-Albrechts Universität (Kiel)
Oldenburg Universität (OLD)

1 Introduction

Axiomatic semantics definitions with so-called "laws of programming" has been advocated as being a natural method that is easy to use for verification without requiring deep specialized mathematical knowledge (see Hoare et al. [HHJ+87]). The appealing feature of this method is that the semantics of a language is described only by equations (or refinement relationships) relating syntactic objects of the language without explicit mention of semantic entities.

Reasoning based on algebraic specification works in a similar way, and so it is a quite obvious idea to use such techniques to support verification employing "laws". Following Hoare [Hoa91], compiler development in the ProCoS project has become an area where this style of reasoning is an accepted technique, and Sampaio [Sam93] describes a method to provide automated verification support for a special approach to compiler design (the normal-form approach, see Hoare, He, and Sampaio [HHS93]). The tool used in this work is the OBJ3 system [GW88].

A different approach to compiler design in presence of real-time constraints is proposed by Müller-Olm [MO95]. Instead of using algebraic laws to transform source programs into programs whose structure is close to machine code as in [HHS93], he defines different levels of abstraction from machine code in order to be able to compare source and compiled program. He does share with [HHS93], however, the method of defining the target code in terms of the source language.

In this paper, we will describe how Müller-Olm's approach can be supported with the Prototype Verification System (PVS, see [Rus93]). This specification and verification system is a state-of-the-art tool incorporating a very expressive higher-order specification language and powerful methods to control proofs. We will not go into much detail of problem modelling and actual proof management, but rather present the underlying philosophy.[1] The main emphasis in this paper will be on development of problem-specific strategies, since this is (a) of utmost importance for successful application of a proof tool, and (b) rather effectively supported by PVS.

We start with a short introduction to compiler development in the style of [MO95], followed by general considerations about verification support issues. In particular, this section gives reasons why we have chosen to work with PVS. The next section deals with fundamental problems of the algebraic semantics definition technique; it describes ways to keep explicit mention of semantic entities outside of the modelling. Section 5 then presents a summary of our experiences with PVS, regarding both specification and verification aspects.

2 An Algebraic Approach to Compiler Design

The approach to compiler design upon which this work is based is used by Müller-Olm in [MO95] and goes back to work presented by Hoare [Hoa91]. Its main characteristics are the following:

- The target language is described in terms of the source language.
- Algebraic laws are defined that allow reasoning with source language programs.
- The correctness notion is based on a refinement ordering $\sqsubseteq$ on programs.
- Using the algebraic laws, source programs are transformed into (source program representations of) target programs that are "better" w.r.t. $\sqsubseteq$.

[1] A more complete account containing the full specification and full proofs can be found in Meyer [Mey95].

2.1 The Process Language

In order to make the approach work more smoothly, the source language (or "reasoning language", as it is called by Sampaio [Sam93]) is extended by specification constructs that do not correspond to implementable programs. In particular, assumptions and assertions about program states are integrated into the language.

In our case, the source language of so-called processes is designed for a real-time context (see Bowen et al. [BHH+94]). It contains the following constructs:

- Π, the "skip" process that terminates immediately without performing any action.
- Ω, the "stop" process that neither terminates nor communicates.
- $\perp$, the worst process (w.r.t. $\sqsubseteq$) that does not satisfy any specification ("chaos").
- $\top$, the best process (w.r.t. $\sqsubseteq$) that satisfies every specification ("miracle").
- $x := e$, the parallel assignment process that terminates immediately. Any of the right-hand side expressions may be $\perp$ meaning that the contents of the corresponding variable becomes undefined after the assignment.
- $c\,!\,e$ and $c\,?\,x$, immediate output to and input from the synchronous channel c.
- $P\,;\,Q$, sequential composition of the processes P and Q.
- $P \lhd e \rhd Q$, the conditional process also written `if` e `then` P `else` Q.
- $e^{\top}$, the assumption of e, defined as $\Pi \lhd e \rhd \top$.
- $e^{\perp}$, the assertion of e, defined as $\Pi \lhd e \rhd \perp$.
- x^{+} and x^{-}, adding variable x to and deleting it from the state space.
- $\Delta\, k$, an active delay of at most $k \in \mathbb{N}$ clock cycles.

A distinctive feature of this language is that none of the atomic processes and none of the process constructors consume any time except for the delay process. So all time consumption in processes becomes explicit.

In [MO95], the semantics of this language is defined axiomatically by means of a set of algebraic laws in the form of equations or refinement relationships, e. g.

Law (empty-assign) $\quad x := x \;=\; \Pi$

Law (variable-initialization) $\quad x^{+}\,;\, x := e \;\sqsupseteq\; x^{+}$

There are a total of 60 laws of this kind. To eliminate doubts about the consistency of such a large system, Müller-Olm has proved that certain predicate transformers provide a model for this system[2].

[2] This proof is going to be part of his forthcoming doctoral dissertation [MO]

2.2 Abstracting from Machine Details

In order to derive a verified compiler, Müller-Olm first defines the effect of target machine (in this case transputer) instructions by simple source language processes, consisting mainly of delays indicating time consumption (as taken from the reference manual [Inm88]), assignments to variables representing the machine state, and some assertions. As an example, let us look at the definition of the effect of "jump" instructions:

$$E_0(\mathtt{j}) \sqsupseteq \Delta\, 3 \ ; \ \mathtt{IP}, \mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{OReg} := \mathtt{IP} + \mathtt{OReg}, \bot, \bot, \bot, 0 \tag{1}$$

This refinement relationship expresses that one may assume at least ($\sqsupseteq$) the following about the basic effect (E_0) of a jump instruction (whose mnemonic is `j`): It takes up to 3 clock cycles, adds the contents of the operand register to the instruction pointer, makes the registers `A`, `B`, and `C` undefined, and clears the operand register.

After this machine level has been defined, abstractions are introduced that correspond to increasingly abstract assembler languages. This is motivated by the wish to be able to compare source processes with their compiled versions in the refinement ordering. Source processes, however, do not mention the machine state variables, e. g. , so the machine dependent elements have to be hidden by appropriate abstractions before the comparison can take place. This abstraction is not performed in one single step in order to be able to deal with different problems in different passes.

The following abstractions are defined:

- First, the control flow is described symbolically. This results in a description of machine instructions that not explicitly mention the instruction pointer anymore.
- The second step abstracts from the operand register, allowing to understand sequences of `pfix` and `nfix` instructions together with the following instruction as one large multi-byte instruction.
- The third step partitions the transputer memory into an indexed set of workspace variables, getting rid of the need to mention the whole memory in every store instruction.
- Fourth, source language variables are represented by workspace locations by making use of a symbol table. This is the first abstraction step that depends on the source language.
- Finally, the machine registers are hidden.

2.3 Compiler Correctness

Now that these five abstractions E_1 to E_5 have been defined, it is possible to set up a correctness predicate for the compiler. A simple version of this predicate not considering timing aspects is the following: A machine program m implements a source process P iff

$$I_5\, a \bullet m \cdot b \ \sqsupseteq \ P \ ; \ I_5\, a \cdot m \bullet b \quad \text{for all instruction sequences } a \text{ and } b. \tag{2}$$

Here, $a \cdot b$ is the concatenation of code sequences a and b, and $I_5\, a \bullet b$ describes the effect of loading the program $a \cdot b$ and starting it at the beginning of b, taking into account the abstractions E_1 to E_5.

Considering timing aspects, the above predicate is too restrictive, however, so a more elaborate version is actually used by Müller-Olm. Yet, the basic idea remains the same. For details, see [MO95].

The problem to prove (2) for arbitrary processes P is a modular one. So it suffices to consider each atomic process and each process constructor in isolation. Thanks to this structure (and some cleverly inserted lemmas), most of the proofs in [MO95] are fairly short. On the other hand, however, a large number of these proofs are necessary since, in order to prove (2), similarly structured properties have to be proved on each abstraction level.

Another approach to compiler design based on similar ideas but not including the abstraction levels is presented by Hoare, He, and Sampaio in [HHS93]. Here, the idea is to use algebraic laws to transform source programs into a kind of normal form that closely corresponds to the execution cycle of the target machine. Such normal form can then be transformed mechanically directly into hardware. In contrast to [MO95], this approach does not consider real-time aspects of the code. A mechanization of proofs of the kind appearing in [HHS93] has been produced by Sampaio [Sam93] using the term rewriting facilities of the OBJ3 system [GW88].

3 Verification Support

As mentioned above, the verification problem in [MO95] consists of a large number of proofs, each having only modest size. Fortunately, many of these proofs can be grouped together in classes since they differ only slightly, e. g. only when it comes to apply the semantics definition for the target instruction occurring in a particular proof. In hand proofs, the normal procedure for such cases is to explicitly perform the proof for some few cases, and to discard the rest of the proofs of the same class as being "analogous".

This is one of the situations where application of proof support tools fits best. Such tools usually provide some method for "replaying" proofs that have been tried before (possibly with modifications), and so there is a very good point in letting the tool calculate the tedious proofs the proof developer herself hadn't bothered to finish explicitly. In this way, the problem of proving with false analogies disappears, since each proof can be checked more or less (depending on the tool and on the particular situation) automatically.

So we have a reasonable starting point for our work. We are not so much interested in applying automated theorem provers; rather, we want to use specification and verification tools to check work already done (in the above sense) by hand. In our special case here, for most theorems and lemmas to be proved there is at least an outline of the proof structure in [MO95] that we can build upon.

If we want to achieve our goal using a particular proof tool, this tool must meet some basic requirements:

- The mathematical background of the problem must be easy to represent within the tool's specification formalism. The tool should enforce as few notational restrictions on the user as possible. Every syntactic transformation of the formulation of a problem makes it less easier to see that original and final transformed version really mean the same.

  Of course, it is even more problematic if semantic notions cannot be expressed directly, so the tool should have a sufficiently expressive specification language.

- The tool must provide methods to store (parts of) proofs, to replay these, and, better still, to define problem-specific strategies.
- The tool must offer ways to visualize the current proof state, be it in graphical or in text form. The output during proofs must be comprehensible to the user without detailed knowledge of the tool's internal organization. It should be possible to select the amount of output required in order to skip listing well-understood parts or to focus on critical ones.
- The proof methods supplied by the tool must be powerful enough. If the user is required to start every simple proof step by hand, there is a great danger that he becomes too much distracted for the overall proof organization, and hence the tool loses much of its helpfulness.

  The tool should supply implementations of standard mathematical theories and effective proof methods for them.

3.1 Choosing a Proof Tool

Considering the first of these requirements, the algebraic setting of the problem strongly suggests to use to tool whose specification formalism is based on algebraic specification or at least provides algebraic techniques as one means of specification. These techniques alone, however, are not sufficient; at least full first-order predicate logic should be supported. If this is not the case, specifications might have to use encodings to express certain notions, and this may lead to a less direct style of specification. In [But94], we have given some examples for such encoding where an earlier version of the Larch Prover [GG91] not supporting the free use of quantifiers is employed to reason about quantified formulas.

The are a number of proof tools satisfying the above requirements to some extent, to name but a few OBJ3 [GW88], KIV [HRS90], or LCF descendants such as HOL [Gor88] and Isabelle [Pau90]. Due to our restricted manpower, however, we were not able to examine these systems in detail. Rather, we first made some experiments with the Larch Prover (LP, see [GG89]), and then switched over to use the Prototype Verification System (PVS, see [Rus93]).

LP is a proof tool based on rewriting techniques. Starting with release 3.1, it supports full first-order many-sorted predicate logic. In another context [But94], we have already worked a lot with former versions of the system. During that work, LP has proved to be easy to use; in Martin and Wing [MW93], some more applications are described.

By applying a special frontend, we have also demonstrated that the restriction to first-order logic can be alleviated to some extent; see Chapter 4 of [But94] for details. Using this frontend, one can also simulate definitions of abstract data types (ADTs) with initial semantics [Wir90], a concept that is missing in LP with its loose semantics for algebraic specifications.

The main reason for not sticking to LP was that two important concepts are not supported (at least up to now), viz. the definition of problem-specific proof strategies and a useful set of built-in sorts with corresponding laws and proof methods. In LP, strategies can only be mimicked by storing sequences of proof commands (so-called "proof scripts") in files and reloading these files afterwards. It is not possible, however, to supply these scripts with parameters. Moreover, the control flow in these scripts is strictly linear, so no "programmed proofs" are possible that use branching to skip over parts of proofs. In the light of the considerations at the beginning of this section, this is a substantial drawback of LP.

Built-in sorts are particularly important for our problem since the real-time aspects of compiler correctness call for simple ways of calculating with natural numbers. In LP, such numbers have to be defined in the usual "algebraic way", e. g. as a sort generated by 0 and the successor operator. Such a representation, however, makes it very hard to perform effective calculations and to present their result to the user in a comprehensible form: Even if `3` and `5` are introduced as shortcuts for `s(s(s(0)))` and `s(s(s(s(s(0)))))`, the result of evaluating the term `3+5` will still be printed as `s(s(s(s(s(s(s(s(0)))))))).`

PVS is a proof tool that does not exhibit these drawbacks. Strategies may be defined using the full expressive power of Common Lisp [Ste90], and several sorts of numbers together with decision procedures for these are included. PVS's logic is a many-sorted full higher-order logic. Instead of using a rewriting-based proof notion, it is based on a sequent calculus; a large variety of basic proof strategies are provided.

Among the useful features of the specification language not available in LP are parameterized theories and predicate subtypes in the style of `PosNat : TYPE = {n : Nat | n > 0}`. Algebraic specification is supported in full extent. Due to the more expressive logic, specifications can often be written more concisely, and many of the necessary consistency checks can be left to the type checker.

3.1.1 Sequent Notation and Proof Trees in PVS

Sequents occurring in PVS proofs are written in the following form:

```
{-1}  A_1
{-2}  A_2
[-3]  A_3
        ⋮
|-------
{1}   B_1
[2]   B_2
        ⋮
```

The A_i and B_j are called *sequent formulas*; the A_i are the *antecedent formulas* and the B_j the *consequent formulas*. Both parts may be empty. The interpretation of a sequent is that the conjunction of the antecedents implies the disjunction of the consequents: $(A_1 \wedge A_2 \wedge \ldots) \Rightarrow (B_1 \vee B_2 \vee \ldots)$. Proofs are maintained in the form of *proof trees*. When representing such trees, we will label their nodes with sequents or proof commands in the following style:

```
      {-1} formula1
        |-------
      [1] formula2

    proof-command1

[-1] formula1
   |-------            |-------
{1} formula4      {1} formula5

             proof-command2

          |-------          |-------
      {1} formula6      {1} formuala7
```

This tree is interpreted in the following way: `proof-command1` generates two subgoals from the topmost sequent, `proof-command2` again generates two subgoals from the second of these, and so on. Formulas whose number is in {} have been affected by the last proof step, and those whose number is in `[]` are unchanged.

4 Levels of Reasoning: Syntax vs. Semantics

4.1 Two Types of Equality

Most of the laws in [MO95] are simple equations or refinement relationships that can easily be modelled as axioms with any proof tool based on some form of algebraic specification. In the following, we will use PVS as an example for such a tool and use PVS syntax. The problems described, however, also apply to other tools such as LP or OBJ3.

As a first example for a law about processes, consider the following:

Law (Π-;-unit) $\Pi \;;\; P \;=\; P$

In order to represent this law, we first have to introduce the set of processes the variable P ranges over and to define the representation of the operators:

```
Proc : TYPE
seq : [Proc, Proc -> Proc]
```

PVS does not allow to define an infix operator `;`, so we have chosen the prefix operator `seq` instead.[3] If we now add the definition of the skip process (`skip : Proc`), we can formulate the above law in the following form:

```
skip-unit : AXIOM FORALL (p: Proc): seq (skip, p) = p
```

Similarly:

Law (scope-simulation) $x^- \;;\; x^+ \;\sqsubseteq\; \Pi$

[3] For more on the problems of this representation, see Section 5.2 below.

requires the definition of the set of identifiers and the begin and end processes for identifiers:

```
Idf : TYPE
beginp, endp : [Idf -> Proc]
```

With these definitions, the law is represented as

```
scope-sim : AXIOM FORALL (x : Idf): seq (endp(x), beginp(x)) <= skip
```

where

```
<= : [Proc, Proc -> Bool]
```

is the representation of the refinement relation. This operator will be used as an infix operator.

Some of the laws, however, have **side conditions** which are less obvious to model and which bring up the fundamental question what the built-in equality operator of the proof tool is to mean. Consider the following law:

Law (scope) $x^+ \; ; P \; = \; P \; ; x^+ \;,$ if P does not refer to x

Modelling this law in a top-down fashion, we could start by defining an operator

```
notin : [Idf, Proc -> Bool]
```

with the intended meaning: `notin(`x`, `p`)` $\equiv$ "p does not refer to x". Next, we could specify the conditional equation

```
scope : AXIOM FORALL (x: Idf, p: Proc):
        notin (x, p) => (seq (beginp(x), p) = seq (p, beginp(x)))
```

The `notin` operator could then be defined in the usual structurally inductive style, with axioms of the following kind:

```
notin_skip : AXIOM FORALL (x: Idf): notin(x, skip) = true
notin_seq  : AXIOM FORALL (x: idf; p1, p2: Proc) :
             notin(x, seq(p1, p2)) = (notin(x, p1) & notin(x, p2))
```

Note that this approach requires to have a `notin` operator for each syntactic category (thus making it desirable to use a tool allowing operator overloading such as LP or PVS). Together with the defining rules, however, these operators could be generated automatically from the definition of the syntactic domains in the style we have described in Chapter 4 of [But94]. Hence these operators do not create a huge definitional overhead.

The problem with this style of definition is that it produces an inconsistent theory as can be seen with the following rewriting proof: Let `x` and `y` be distinct objects of type `Idf`. Applying equations defined in the style described above, we can generate the following rewritings:

```
                    notin(y, x:=y;x:=1)
                   /                    \ (*)
                  v                      v
notin(y, x:=y) & notin(y, x:=1)      notin(y, x:=1[y/z])
                |                            |
                v                            v
notin(y, x) & notin(y, y) & ...       notin(y, x:=1)
                |                            |
                v                            v
       ... & false & ...             notin(y, x) & notin(y, 1)
                |                            |
                v                            v
              false                     true & true
                                             |
                                             v
                                            true
```

So `true = false` has been proved, and hence the theory has been shown to be inconsistent. The source of the problem is the step labelled with (*); all the other steps result from applying rules defining the `notin` operators, but this step makes use of one of the laws for processes

Law (combine-assign) $\quad x := e \; ; \; x := f \quad = \quad x := f[e/x]$

(resp. its representation). This, however, is a *semantic* law describing when two syntactically different processes have the same effect. The rules defining the `notin` operators, on the other hand, specify a *syntactic* property. Obviously, it is incorrect (in the meta-level sense) to reason about syntactic properties of objects by considering a semantically, but not syntactically, equivalent object; thus the rewriting step (*) should not have been allowed.

So mixing syntactic and semantic laws is the source of the problem, and hence a solution is to keep these rules apart. This can be achieved in two rather different ways:

(1) **The syntactic approach**

Equality between syntactic objects is looked upon as *syntactic equality*. This means that the law "combine-assign" is not available in the form given above, and hence the rewriting step (*) is not possible.

In order to state the semantic laws, this approach requires to define a new operator interpreted as *semantic equality* for each syntactic domain, e. g.

```
== : [Proc, Proc -> Bool]
```

Using these operators, all the semantic laws must be rewritten.

(2) **The semantic (or pragmatic) approach**

Equalities continue to mean *semantic equivalence*. In order to overcome the inconsistency problem, one guarantees by meta-level operations that syntactic and semantic rules can never be mixed.

Using LP, such an operation could be selective activation/passivation of rules. With PVS, the mechanism could be equivalently controlled using the list of auto-rewrite rules that are applied without user interaction.

In accordance with Sampaio [Sam93], we have chosen the second approach because of the following reasons:

- The semantic approach coincides with the usual mathematical style of reasoning that also keeps syntactic and semantic calculations apart.
- To be useful, the new semantic equality operators must exhibit all the distinctive properties of equality. In particular, they must be defined as congruence relations with respect to all occurring operators. To achieve this, a lot of rules have to be defined, thus making the prover less efficient.
- Built-in proof methods supporting and relying on equality would not work for the newly-defined semantic equality and would have to be replaced by problem-specific ones. Since the built-in methods are often defined in a low-level style close to the core of the prover and since this style of definition is often not available for the user, the replacing methods would most probably be less efficient and useful than the original ones.
- There is no danger for mistaking one equality operator for the other.

So we have two disjoint sets of equations: those defining syntactic properties and operations such as (non-)occurrence and substitution, and those defining the semantic operations.

4.2 Syntactic Implication

Another problematic side condition shows up in the following law:

Law (exploit-assumption) $e^\top \; ; \; x := f \;\; = \;\; e^\top \; ; \; x := g \;,$ if e implies $f = g$

Here we see that the syntactic entities e, f and g (standing for expressions) occur in a semantic context inside the side condition. Strictly speaking, the Boolean expression e cannot "imply" anything; only its semantics can, given an environment assigning values to the identifiers contained. Similarly, the equality $f = g$ is to be seen semantically.

Two ways to handle this situation are possible:

The first is to explicitly define a semantics function for expressions, depending on some sort of environments, e. g. mappings from identifiers to values:

```
sem : [Expr, Env -> Value]
```

A reasonable choice for the set of values in our case would be the set of integers representable as a word in the transputer memory. Similarly, a semantics function for Boolean expressions is defined[4]:

```
sem : [BExpr, Env -> Bool]
```

Using these functions, the side condition from above can be specified as

```
FORALL (env : Env): sem(e,env) => (sem(f,env) = sem(g,env))
```

[4] Again note that a convenient modelling relies on overloading.

The advantage of this solution is that it does not require any new definitions except for the semantics functions. Moreover, syntax and semantics are clearly separated, avoiding a possible source of confusion.

The major disadvantage is the introduction of an operator (`sem`) and a concept (environments) that are both left implicit in the original problem description. Moreover, explicit mention of semantics does not quite accord to the spirit of the overall approach of only reasoning with syntactic objects.[5] Therefore, we do not make use of this solution.

The second way of handling the situation is to leave the semantics implicit as in the original. In order to be able to express the side condition from above, we define a "syntactic application" operator:

```
=> : [BExpr, Bool -> Bool]
```

Together with the operator, we set up a set of laws defining its properties, e. g.

```
FORALL (a, b:  Expr):  eq(a,b) => a = b
```
(3)

where `eq :  [Expr, Expr -> BExpr]` is the equality operator of the source language.

Of course, these laws must be verified against the semantics of expressions. Since this semantics is left implicit, however, this cannot be done within the tool's formalism; rather, they must be checked using meta-level reasoning. This is very similar to the verification problem for the laws about processes, and it shows that the introduction of this syntactic implication conforms with the spirit of the approach: Properties of syntactic objects are expressed using only syntactic laws (including semantic equality, see the previous section), and verification of these laws is left to the meta level.

Note that it is not necessary to define a complete theory of syntactic implication. Rather, it suffices to specify just those laws that are needed for the proofs at hand.[6] Regarding the proofs in [MO95], this means that only very few laws are actually needed, and all of them of a similar complexity as that of (3) above.

5 Experiences With PVS

5.1 Specification Aspects

As said before, most of the setting in [MO95] consists of relatively simple algebraic laws. So, after having sorted out the conceptual difficulties mentioned in the previous section, we did not have fundamental problems to write an algebraic specification of the problem. And due to the very expressive specification language of PVS, implementing this specification with PVS was rather straightforward (for details, see Meyer [Mey95]).

Some features deserving explicit mention are the following:

[5]Of course, one can argue that the law "exploit-assumption" itself violates this spirit since the reference to semantics is merely disguised in this law.

[6]This depends on the proof tool employing loose specification which is true e. g. for LP and PVS.

- Using **parameterized theories**, specifications can become more concise and easier to handle. In our case, we use lists of various kinds (e. g. of instructions or byte expressions), and by abstracting from the element type, we can save specification as well as proof effort, since general properties can be proved for the abstract data type once, continuing to hold for instances.
- **Higher-order concepts** can make specifications more readable. Take for example the problem of introducing a preorder, i. e. a reflexive and transitive relation. With a first-order tool such as LP, one would have to define the corresponding operator and then the laws defining its properties, e. g.

  ```
  declare operator __ <= __ : Proc, Proc -> Proc
  assert
    % \A is the LP notation for the universal quantifier
    :ref_refl:  \A p: Proc (p <= p)
    :ref_trans: \A p1: Proc \A p2: Proc \A p3: Proc
                ( (p1 <= p2) /\ (p2 <= p3) => (p1 <= p3) )
  ..
  ```

  With PVS, we can profit from higher-order logic and from the large set of predefined theories. In our example, we can build on top of the theory defining properties of relations can can write

  ```
  IMPORTING relations [Proc]   % parameterized theory
  <= : [Proc, Proc -> Bool]
  refPreOrder : AXIOM preorder? (<=)
  ```

  The functionality of `preorder?` is here `[[Proc, Proc -> Bool] -> Bool]`. Besides being clearer due to being shorter and to the use of self-describing names, this form of specification has the advantage that there is no need to invent axiom names for every new preorder: the `IMPORTING` mechanism generates standard names for all instances of the `relations` theory, if necessary.
- **Predicate subtypes** allow to specify in a way that comes close to order-sorted specification as e. g. in OBJ3 (see Goguen and Meseguer [GM89]). As an example, consider the case of "word expressions", i. e. expressions whose value can be represented in a machine word (32 bits). This type has the subtype of "byte expressions" (8 bits), and this type again has the subtype of "nibble expressions" (4 bits). In OBJ3, one would write something like

  ```
  sorts NibbleExpr ByteExpr WordExpr .
  subsorts NibbleExpr < ByteExpr < WordExpr .
  ```

  Using predicate subtypes, the same situation is modelled in PVS as

  ```
  WordExpr : TYPE+              % non-empty type
  byte? : pred [WordExpr]
  ByteExpr : TYPE = (byte?)     % those words that satisfy byte?
  nibble? : pred [ByteExpr]
  NibbleExpr : TYPE = (nibble?) % those bytes that satisfy nibble?
  ```

> Of course the OBJ3 version is much more suggestive; nevertheless the PVS version has proved to be useful to express the relations between the different sorts of expressions.[7]

So the semantic aspects of specification with PVS are quite satisfactory. The syntactic side, however, could do with some improvements. The problem is that the set of infix operators is rather restricted, and that "mixfix" operators such the usual *if-then-else-fi* cannot be defined at all. Examples for tools that are more flexible are OBJ3 or the ASF+SDF Meta-environment (see van Deursen [vD94]), allowing arbitrary mixfix definitions. LP does not support full mixfix notation, but the set of infix operators is fairly large.

As as example where syntactic restrictions get into the way, consider the sequential composition operator on processes in [MO95]. This operator is associative, and hence one can write $p_1 ; p_2 ; p_3$ without having to care about precedences or parentheses.

PVS does not allow an infix operator `;`, however, and on the other hand, no legal infix operator name seems intuitive either. So the typical solution is to define a binary prefix operator, say,

```
seq : [Proc, Proc -> Proc]
```

that models sequential composition. But then one has to write parentheses, and one has to decide between `seq (p1, seq (p2, p3))` and `seq (seq (p1, p2), p3)`. This is particularly inconvenient if one wants to reason about such terms. Both forms can be transformed into each other easily by applying the associativity law, but these transformations become tiresome if they every now and then just because some term structure does not match. (See the next section for a discussion of problem-specific strategies that alleviate this problem to some extent.)

As shown by Skakkebæk [Ska94], these restrictions can be overcome by building a new frontend and backend on top of PVS. This solution, however, requires additional programming. It would be much more convenient to have syntactic expressiveness in PVS without having to modify or extend it.

The point is that syntax is not just "sugar". It can make users feel comfortable with a proof tool, but it can also prevent them from using it from the outset. The more an original problem specification has to be transformed for use within a proof tool (syntactically or even semantically), the less the tool is going to be accepted and the less likely it is going to be used seriously.

5.2 Verification Aspects

One of the most prominent features of PVS is the large set of decision procedures that are built into the system. Quite often, they allow to prove simple goals without further interaction. This is a great advantage over tools that require the user to initiate every single step manually, since it frees her from considering every tiny detail of a proof and lets her concentrate more on proof structuring.

Another important aspect is the possibility to define problem-specific strategies. Every problem field has its own special proof methods, built from simpler methods by applying them in some

[7] Note that in both cases, the subtypes are not interpreted in the above sense; there is no connection yet to lengths measured in bits. This interpretation has to be added by additional axioms or definitions of the recognizer functions.

fixed order. If a proof tool does not support such methods (e. g. by supporting the definition of custom strategies), the simple parts of the methods have to be applied in the right order under manual control. As a result, the really difficult parts of proofs – where the more complex methods are selected – are in danger of getting lost in the ever-the-same small details. This leads to less effective usage of the tool since it becomes harder to keep an overview of the proof, and hence the probability of false decisions (errors) increases. Moreover, proofs become less readable if every simple detail is always spelt out in its entirety. Both problems are very likely to reduce the user's acceptance of the tool significantly: one doesn't like a tool that makes it difficult to finish proofs successfully and whose output is hard to understand.

As a general rule, a proof tool should not dictate proof layout more than absolutely necessary. To achieve this goal, there must be some way of defining problem-specific strategies.

Still, not very much is gained if strategy definition itself is again a painful and complicated process. It is also important that the strategy definition language be expressive and flexible enough to allow straightforward formulation even of complex algorithms to prove a goal.

In PVS, this condition is fulfilled since the "proof implementation language" is full Common Lisp [Ste90]. As a safety feature, however, it is only possible to change the proof state (including the current set of goals) by applying one of the primitive predefined strategies. So the correctness of the system cannot be compromised by adding faulty strategies.

The current PVS version (2.175 alpha) does not provide significant documentation regarding strategy definition. Moreover, debugging of strategies is rather hard since the only means to do so is to use the standard Lisp debugger (which does not know about strategies). Yet, we have been able to implement some strategies that support proofs performed in the style of Müller-Olm [MO95]. Although these strategies are still fairly basic regarding their meta-level complexity, they already consist of a considerable amount of Lisp code. In the following, we will discuss some of their characteristics.

As examples for special strategies, we will consider parts of the following:

- a strategy for implementing chains of equations resp. chains of refinement relationships (`sub-le` and `sub-eq`);
- a strategy that models use of monotonicity properties with respect to the refinement relation (`monseq`);
- a strategy for rewriting in presence of an associative operator (`rewrite/replace-assoc`).

5.2.1 Defensive Programming

An imperative that all strategies (be they user-defined or primitive) must adhere to is that the user always shall communicate with the PVS system only and never directly with the underlying Lisp system itself. As a direct consequence, execution of the Lisp code corresponding to strategies must never result in a runtime error, since such errors invoke the debugger in which case the direct contact with the PVS system is lost. Therefore, strategies must be programmed extremely defensively, particularly since Lisp is virtually type-free, and mismatches between expected and actual types of arguments are mostly not reported until runtime. Problems should be detected by the strategies themselves and not by the Lisp system. In this way, there is a chance to generate meaningful error messages and to prevent PVS from entering debug mode.

For our strategies, this has resulted in a very regular pattern of definition. As a first example, consider the strategy `sub-eq` that models reasoning by chains of equations in the following fashion: A proof of the form

$$
\begin{array}{lll}
 & a & \\
= & & \text{[reasons why the first equation holds]} \\
 & b & \\
= & & \text{[reasons for the second equation]} \\
 & c & \\
= & & \text{[reasons for the third equation]} \\
 & d &
\end{array}
$$

is represented in PVS using this strategy as

```
              |-------
           [1] a = d

         (sub-eq "b")

     |-------            |-------
 {1}   a = b        {1}   b = d

                   (sub-eq "c")

                |-------            |-------
            {1}   b = c        {1}   c = d
```

where the proofs of the "n-th equations" have to be filled in at the leaves of this tree. The only difference between this proof tree and the proof performed by hand is that the former always records the equation that is still to be proved.

The definition of this strategy is structured as follows:

```
(defstep sub-eq (step &optional (fnum 1))
  (let ( (fnum-ok?    (when (and (integerp fnum) (> fnum 0)) fnum))
         (fnum-error  (when (not fnum-ok?) ... ))
         ;; ... an error string for illegal formula numbers
         ...
         (start       (when fnum-ok? (sel-left-hand-side fnum)))   ;; -> a
         (left-error  (when (not start) ... ))
         ;; ... an error string for problems with the left-hand side
         (end         (when start (sel-right-hand-side fnum)))     ;; -> b
         (right-error (when (not end) ... ))
         ;; ... an error string for problems with the right-hand side
         (step-expr   (typecheck (pc-parse step 'expr)
                                 :expected (type start)))          ;; -> c
         (first-step  (when step-expr (make-equation start step-expr)))
```

```
15                                                                     ;; -> a = c
16            (second-step (when step-expr (make-equation step-expr end)))
17                                                                     ;; -> c = b
18            )
19       (if step-expr  ;; step-expr <> NIL <=> no error while evaluating arguments
20           ... ;; start the strategy proper
21           ;; Otherwise output the appropriate error message:
22           (if (and start end)
23               (skip-msg "Wrong type in intermediate step.")
24            (if start
25               (skip-msg right-error)
26            (if fnum-ok?
27               (skip-msg left-error)
28               (skip-msg fnum-error)
29               )))))
30      ...
31     )
```

Apart from the term forming the intermediate `step`, the strategy has one more optional parameter `fnum` (line 1). It contains the number of that sequent formula into which `step` is to be inserted. This only makes sense for succedent formulas having a positive formula numbers (lines 2 to 4); the default value is 1.

The form of the body of the strategy is a `let` expression. It is evaluated sequentially; during this evaluation, the actual parameters are checked for plausible values (lines 2 and 12) and the proof state is accessed in small steps in order to be able to generate useful error messages (lines 6 to 11). If all selections have been performed successfully, the intermediate equations are generated (lines 14 to 17), and the strategy proper is started. Otherwise, the appropriate error message is output using the "do-nothing" strategy `skip-msg`.

5.2.2 Global Variables

Rather than using parameter passing, PVS keeps all information about the current proof in a global variable (`*ps*` for "proof state"). The internal structure of this variable's data structure is not documented; all access to its contents has to take place via special selector and recognizer functions (for read access) or via the predefined strategies (for write access). In this way, faulty additional strategies cannot corrupt the system's consistency.

The special strategies for our problem also make use of global variables, but mainly for reasons of efficiency and readability and not for information hiding. Instead of passing commonly used data to every strategy and auxiliary function, these data are kept in special variables. As an example, consider the strategy `monseq` that applies the laws of monotonicity of sequential composition of processes with respect to the refinement relation:

```
RefMonSeqLeft : AXIOM
   FORALL (p1, p2, p: Proc): (p1 <= p2) => (seq (p1, p) <= seq (p2, p))
RefMonSeqRight : AXIOM
```

```
        FORALL (p1, p2, p: Proc): (p1 <= p2) => (seq (p, p1) <= seq (p, p2))
```

monseq deduces information from the current sequent using these laws in the following style:

```
        [-1] a <= b
          |-------
        [1]  p
              ↓
 (monseq -1 "c" :left? T)
              ↓

        {-1} seq (a, c) <= seq (b, c)
        [-2] a <= b
          |-------
        [1]  p
```

In order to generalize the strategy definition from a special setting, we keep some parameters in global variables:

- `*pre-order-set*` contains the type on which the refinement relation is defined (`Proc` in our case).
- `*pre-order-relation*` contains the refinement relation itself (which should be a preorder on `*pre-order-set*`; in our case, it is `<=`).
- `*left/right-mono-seq-axiom*` contain the names of the monotonicity axioms (in our case `RefMonSeqLeft/Right`).

The global settings are used as defaults for optional parameters of the strategy; the complete parameter list is the following:

```
(monseq fnum context
        &optional (left? T) rel_name type_name left_ax right_ax)
```

`left?` determines which of the two monotonicity axioms to use. `fnum` is the number of the target formula for the strategy (must be a negative integer), and `context` is the left resp. right context inserted into the `seq` term (represented by the variable named `p` in the above axioms).

By supplying values for the optional parameters in the style of the example (`:`⟨ *name* ⟩ ⟨ *value* ⟩), the default values can be overridden.

5.2.3 Associativity

The strategies that support rewriting under associativity also make use of global variables; not for storing default values, however, but for storing information about the current goal in a more accessible form than as part of the `*ps*` variable.

The idea of this strategy is quite simple: Sequential composition of processes is associative, so there must be a law of the following kind:

```
SeqAssoc: AXIOM FORALL (p1, p2, p3: Proc):
          seq (seq (p1, p2), p3) = seq (p1, seq (p2, p3))
```

Usual mathematical reasoning constantly uses this law in an implicit way; instead of `seq (seq (p1, p2), p3)`, one writes $p_1 ; p_2 ; p_3$. For PVS, however, `seq` is a binary operator. Hence e. g. the axiom

```
Ax : AXIOM seq (a, b) = c
```

(for `a, b, c: Proc`) cannot be applied directly to the term `seq (a, seq (b, d))`. Instead, one first has to explicitly rewrite it with `SeqAssoc` (from right to left) in order to generate the term structure required in axiom `Ax`.

The strategy `rewrite-assoc` generalizes the predefined `rewrite` strategy in this respect, e. g.

```
   |-------
[1]  seq (a, seq (b, d)) = e
              |
              v
   (rewrite-assoc "Ax" 1)
              |
              v
        |-------
      {1}  seq (c, d) = e
```

Applying this strategy mainly results in the following steps:

(1) Extract flattened list of `seq` arguments from the target formula (in the example `<a, b, d>`) and from the axiom that is to be applied (in the example `<a, b>`). This information is stored in special global variables in an easily accessible form.

(2) Check whether the `seq` arguments from the axiom form a consecutive sublist of those from the target.

(3) If this is the case: Gradually transform the target formula into the right form using the associativity axiom and the predefined `rewrite` strategy, and finally rewrite the target formula using the axiom.

(4) Otherwise do nothing; the axiom cannot be applied.

A similar strategy `replace-assoc` has been defined as an extension of the predefined strategy `replace`.[8]

Up to now, this strategy can only be used if the axiom is a ground formula (without variables). In order to extend the algorithm to handle general formulas, full associative matching has to be implemented for step (2).

The definition of `rewrite-assoc` turned out to be fairly long and complex, mostly due to the need to extract the list of `seq` arguments in step (1) and the rather indirect way of changing the proof state only with predefined strategies in step (3). PVS would be much more useful for our sort of problem if associative operators were supported on a basic level. Directly using the data structures of the proof state certainly would result in a more efficient solution for the strategy that we could only define at great expense.

As a side remark, it is interesting that LP (in contrast to OBJ3) does not support associative operators either. It does support, however, operators that are both associative *and* commutative.

[8]The main difference between `rewrite` and `replace` is that the former applies rules from the rule database (axioms, lemmas and such), whereas the latter applies equations that are contained in the current sequent.

It seems that operators that are "only" associative do not occur very often outside of sequential programs. (But then, what about list concatenation?)

6 Conclusion

In our experiments, we have found that PVS provides a useful basis for supporting compiler verification in the style of [MO95]. Its specification language is sufficiently expressive to allow a rather direct implementation of the mathematical framework, a problem being that the syntactic restrictions of PVS prevent an intuitive modelling of some infix operators.

In order to be able to use PVS's proof component effectively, one has to develop specific strategies that allow to structure proofs in a way that is similar to usual mathematical reasoning as demonstrated e. g. in [MO95]. Development of such strategies is a considerable effort, but it is the only way to allow the user to abstract at least from some of the proof details and to concentrate on more global proof design.

We did not want to use PVS as an automated theorem prover for the logic in [MO95], but rather as a proof checker that is able to replay proofs based on a given proof layout. For this purpose, it is absolutely important to be able to take proof steps with PVS whose complexity is similar to that of manual proof steps, because otherwise the connection between original proof and replayed proof may become unclear. Our first results with more complex strategies have shown that it is possible to define strategies that permit such complex proof steps. Since the current PVS version is still at the alpha test stage, strategy development is somewhat painful due to missing documentation of system internals, but this is certainly going to become better with reaching later stages of development.

Sequential composition is one of the most frequently occurring operations in the proofs in [MO95]. Moreover, one of the basic proof techniques there is to move statements within such sequences. Therefore it turned out to be particularly disturbing that PVS does not support associative operators on a basic level. Constant demand for re-arranging the `seq` operators in a sequence of statements distracts from the proof goal proper and makes proofs longer, less clear, and more time-consuming. Although development of strategies that automatically perform this re-arrangement is possible (if at great expense), built-in support for such operators would mean a substantial improvement.

The ultimate goal when developing proof support mechanisms with a tool such as PVS should always be an environment that provides as much of the original formalism and proof methods and as few artificial extensions and restrictions as possible. In such an environment, the developer of an original manual proof should feel comfortable enough to make efficient use of the system. For the compiler verification setting of this paper, PVS shows the potential for serving as a basis for such an environment.

Acknowledgments

I would like to thank Markus Müller-Olm for helpful discussions about his work and its PVS representation. Thies Meyer did most of the practical work reported here. Discussions with Axel Dold and Holger Pfeifer helped clarifying some of the problems mentioned in Section 4.

References

[BHH+94] Jonathan Bowen, C. A. R. Hoare, Michael R. Hansen, Anders P. Ravn, Hans Rischel, Ernst-Rüdiger Olderog, Michael Schenke, Martin Fränzle, Markus Müller-Olm, Jifeng He, and Zheng Jianping. Provably correct systems — FTRTFT'94 tutorial. ProCoS Technical Report COORD JB 7/1, Oxford University Computing Laboratory, September 1994.

[But94] Karl-Heinz Buth. *Techniques for Modelling Structured Operational and Denotational Semantics Definitions with Term Rewriting Systems*. PhD thesis, Mathematisch-Naturwissenschaftliche Fakultät, Christian-Albrechts-Universität Kiel, 1994. Available as Bericht Nr. 9414, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel.

[GG89] Stephen J. Garland and John V. Guttag. An overview of LP, the Larch Prover. In Nachum Dershowitz, editor, *Proceedings of the Third International Conference on Rewriting Techniques and Applications*, LNCS 355, pages 137–155. Springer-Verlag, 1989.

[GG91] Stephen J. Garland and John V. Guttag. *A Guide to LP, The Larch Prover*. Massachussetts Institute of Technology, November 1991. Release 2.2.

[GM89] Joseph A. Goguen and José Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. Technical Monograph PRG-80, Programming Research Group, Oxford University Computing Laboratory, December 1989.

[Gor88] Michael J. C. Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwistle and P.A. Subramanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer, 1988.

[GW88] Joseph A. Goguen and Timothy Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI International, August 1988.

[HHJ+87] C. A. R. Hoare, I. J. Hayes, He Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorensen, J. M. Spivey, and B. A. Sufrin. Laws of programming. *Communications of the ACM* **8**(8), 672–686, August 1987. Corrigenda no. 9, p. 770.

[HHS93] C. A. R. Hoare, He Jifeng, and Augusto Sampaio. Normal form approach to compiler design. *Acta Informatica* **30**, 701–737, 1993.

[Hoa91] C. A. R. Hoare. Refinement algebra proves correctness of compiling specifications. In Carroll C. Morgan and James C. P. Woodcock, editors, *Proceedings of the 3rd Refinement Workshop*, Workshops in Computing Series, pages 33–48. Springer-Verlag, 1991.

[HRS90] Maritta Heisel, Wolfgang Reif, and Werner Stephan. Tactical theorem proving in program verification. In Mark E. Stickel, editor, *Proceedings of the 10th International Conference on Automated Deduction, Kaiserslautern, FRG*, LNAI 449, pages 117–131. Springer-Verlag, July 1990.

[Inm88] Inmos Ltd. *Transputer Instruction Set – A Compiler Writer's Guide*. Prentice-Hall International, 1988.

[Mey95] Thies Meyer. Anwendung von PVS zur Codegenerator-Verifikation. Master's thesis, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel, October 1995. In German; under preparation.

[MO] Markus Müller-Olm. Modular compiler verification (preliminary title). Phd thesis in preparation.

[MO95] Markus Müller-Olm. Structuring code generator correctness proofs by stepwise abstracting the machine language's semantics — Draft —. ProCoS Technical Report Kiel MMO 12/4, Christian-Albrechts-Universität Kiel, August 1995.

[MW93] Ursula H. Martin and Jeannette M. Wing, editors. *Proceedings of the First International Workshop on Larch, Dedham, MA, 1992*, Workshops in Computing Series. Springer-Verlag, 1993.

[Pau90] Larry C. Paulson. Isabelle: the next 700 theorem provers. In P. Oddifreddi, editor, *Logic and Computer Science*, pages 361–385. Academic Press, 1990.

[Rus93] John Rushby. A tutorial on specification and verification using PVS. In Peter Gorm Larsen, editor, *Tutorial Material for FME '93: Industrial-Strength Formal Methods. Proceedings of the First International Symposium of Formal Methods Europe, Odense, Denmark*, pages 357–406, April 1993.

[Sam93] Augusto Sampaio. *An Algebraic Approach to Compiler Design*. PhD thesis, Oxford University, 1993. Available as Technical Monograph PRG-110, Oxford University Computing Laboratory, Programming Research Group, October 1993.

[Ska94] Jens Ulrik Skakkebæk. *A Verification Assistant for a Real-Time Logic*. PhD thesis, Department of Computer Science, Technical University of Denmark, Lyngby, 1994.

[Ste90] Guy L. Steele Jr. *Common Lisp. The Language*. Digital Press, 1990.

[vD94] Arie van Deursen. *Executable Language Definitions: Case Studies and Origin Tracking Techniques*. PhD thesis, Universiteit van Amsterdam, 1994.

[Wir90] Martin Wirsing. Algebraic specifications. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B: Formal Models and Semantics*, chapter 13, pages 675–788. Elsevier/MIT Press, 1990.