Mechanical Software Verification: High Level Control Aspects from a User's Perspective

Wolfgang Goerigk

Christian-Albrechts-Universität zu Kiel, Germany*

Abstract. We present lessons learned from using mechanical theorem proving for proof support in software verification, with trusted execution of programs in mind. We will use two realistic running examples, compiler verification, which is central if we want to prove that we can trust a piece of executable software, and an industrial project in which we proved the correctness of a safety critical expert system using (verified) runtime result verification. We will emphasize the role of partial program correctness and its preservation. And we will comment on high level control aspects, in particular on what we can and what we will not be able to prove for a concrete piece of executable software.

1 Introduction

Our paper is about software verification, *i.e.,* about proving properties of programs which we want to guarantee in order to trust in program execution. But how can we mechanically verify executable machine programs? Will we ever be able to prove non-trivial correctness properties of realistic executable machine programs and furthermore check the proofs using theorem provers? Or should we better transfer such attempts into Wittgenstein's domain of logical scepticism, arguing that no way of reasoning will ever lead towards convincing solutions – hence just give up? Our answer is no. There are solutions. But since we definitely will not succeed to verify large programs semantically on machine code level (at least not in an engineering style), we have to modularize the problem. Hence, our paper is about source level verification of programs and compiler verification.

In spite of the problems mentioned above, there is a high risk either to formulate theorems which are simply not true, or (practically) not provable, or useless. Therefore, a major goal will be to show what we can and what we cannot expect to be able to prove for a realistic piece of executable software. We will emphasize the importance of partial program correctness and its preservation, i.e., of a correctness guarantee for regularly computed non-erroneous results also of machine executables. This is often sufficient, very practical, and sometimes surprisingly easy to prove, even rigorously, formally and mechanically.

We do not believe that the entire work necessary for a convincing software correctness proof can be "push button", *i.e.,* fully mechanical and automatic.

* Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, Olshausenstr. 40, D-24098 Kiel, Germany. Email: wg@informatik.uni-kiel.de

We have to think about what to prove and how to prove it, tasks which are very similar to those proposed to successfully manage software engineering processes. We have to analyze requirements, think about the architecture and the design of proofs, and finally we have to implement them using a particular theorem prover. Additional work is necessary to produce a consistent and complete proof documentation in order to convince our customers and help them to believe both in the formalization and in the proof.

If we do not want to verify programs on machine code level, we have to trust in compilers. Their executable implementations have to guarantee that correctness properties of source programs are preserved, so that we can trust in the correctness of the generated target programs. But note that this opens up a similar question for the compiler and its implementation, which is a piece of (executable) software that we want to trust. In order to avoid circuli vitiosi, *i.e.,* never ending dependencies on unsafe executable software (of uncertain pedigree, [21]), we need for principle reasons some manual proof checking. Fortunately, we can push this effort entirely into compiler implementation correctness proofs and rigorously verify compiler executables to preserve adequate and reasonable correctness properties of source programs[1] [17].

We want to demonstrate the proposed modularization in source level and compiler correctness. For that we use two examples which fit together. In section 2 we sketch an industrial project on the verification of a safety-critical expert system. Its correctness proof has actually been checked by the ACL2 theorem prover [24]. Safety can be proved by a technique which we call (verified) runtime result verification [13]. In section 3 we informally collect requirements for realistic correct compilation. Section 4 will yield a mathematical framework which enables us to precisely define compiler correctness by preservation of source program correctness. In particular, we will sketch a mechanical proof of preservation of partial correctness for a toy compiler in ACL2 (section 5). The role of toy examples is crucial in order to finally succeed, and we will give some comments in section 6. We will end up with some lessons learned and some wishes for theorem provers used in software verification.

2 Verified Runtime Result Verification

The effort of proving the correctness of large software systems seems often not justifiable. Heuristics and programming tricks are necessary in order to to solve complex problems successfully and efficiently. Mathematical induction then often fails because the algorithms to be verified get too complex and tricky. This applies in particular to knowledge-based systems. However, if such systems are safety- or mission-critical, we rely on their correctness.

In [13] we propose a *checker-based* approach to software verification which exploits the idea of *runtime result checking* [4] for verification. It is applicable if

[1] The work reported here has been supported by the Deutsche Forschungsgemeinschaft (DFG) in the *Verifix* and *VerComp* projects on *Correct Compilers* and *Techniques for Compiler Implementation Verification*

partial correctness of the application suffices. Partial correctness can be proved by a-posteriori runtime result verification. Let us sketch an industrial software verification project [3] in which we exploit the technique of verified runtime result verification (checker-based program verification) in order to prove the correctness (soundness) of a safety-critical expert system. The system is called Relais Master [1, 25], and it is an expert system for computing test-plans for relay assembly groups that control railway systems. It is used in the engineering phase of such devices in order to (more automatically) provide support for hardware-in-the-loop tests. The test-plans are generated from circuit descriptions. Later, they are to be automatically executed by a *test-roboter* in order to check the correct electrical behavior of the device.

Although digital circuits get more frequently used today, at least many German railway systems are still controlled by relay blocks. In order to guarantee a sufficiently high level of safety, these devices need regular maintenance, and in particular, a re-certification of every single physical device is required within regular time intervals.

2.1 The Relais Master – A Safety Critical AI Tool

The Relais Master automates the time-consuming and error prone manual test-plan construction. The system has been developed by the German company DTK (Hamburg) in cooperation with the Laboratory for Artificial Intelligence (LKI) at the University of Hamburg, Germany. After scanning the circuit, the engineer transforms it into an internal graph representation, and the system then generates a set of measurements between terminal contacts of the device (the test-plan). Each measurement is augmented with information about the required *state* of relay contacts (*open* or *closed*). The test roboter will later pneumatically switch the relays. We have the following requirements for the generated test-plan:

- It should be *complete* for the circuit, *i.e.,* it should contain every test necessary to detect any combination of any number of defects in the relay group. (*soundness*)
- It should be "good" in the sense that it does not require too many relay switching operations, because the maintenance interval is mainly triggered by the estimated number of switches. (*quality*)

The first property is safety-relevant, whereas, fortunately, the second is not, although it is as important for practical usability and makes the use of AI techniques adequate for the problem. But for safety we only need soundness. In fact, this observation is crucial to our *checker-based* approach: Soundness can quite easily be checked for a given test-plan, whereas the test-plan generation is complicated and has to assure quality as well.

The input of the Relais Master is a circuit description which actually is manually constructed from a scanned circuit plan. Figure 1 shows a very small example and the Ascii representation of the internal object-oriented Relais Master data structure is a Lisp-s-expression of the form:

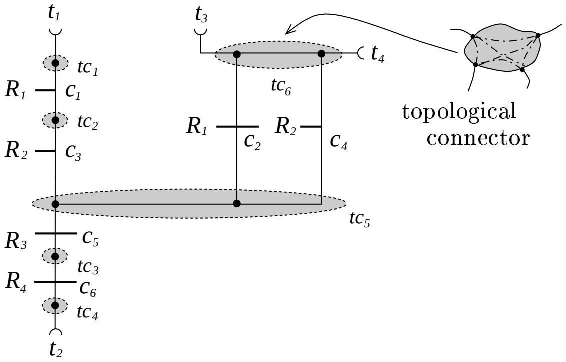

Fig. 1. A sample circuit. Relais R_1 has two contacts, c_1 and c_2, and R_2 has contacts c_3 and c_4. If R_1 is not excited (released), contact c_1 is *closed* (*i.e.,* is to conduct), whereas c_2 is *open* (*i.e.,* is to isolate). Topological connectors are abstractions from wiring and soldering, for n ports, at least $n-1$ connections have to conduct.

(((R_1 released ((c_1 closed) (c_2 open)))
(R_2 released ((c_3 closed) (c_4 closed)))
(R_3 released ((c_5 open))) (R_4 released ((c_6 open)))
t_1 t_2 t_3 t_4)

((t_1 c_1) (t_2 c_6) (t_3 c_2 c_4 t_4) (t_4 t_3 c_2 c_4) (c_1 t_1 c_3) (c_3 c_1 c_5 c_2 c_4)
(c_5 c_6 c_3 c_2 c_4) (c_6 c_5 t_2) (c_2 t_3 t_4 c_4 c_3 c_5) (c_4 t_3 t_4 c_2 c_3 c_5))

((con t_3 t_4 () (R_1 R_2 R_3 R_4)) (con t_3 t_1 () (R_1 R_2 R_3 R_4))
(iso t_2 t_1 (R_3) (R_1 R_2 R_4)) (con t_3 t_2 (R_3 R_4) (R_1 R_2))
(iso t_3 t_2 (R_2 R_3 R_4) (R_1)) (iso t_1 t_2 (R_2 R_3 R_4) (R_1))
(con t_3 t_2 (R_1 R_2 R_3 R_4) ()) (iso t_1 t_2 (R_1 R_3 R_4) (R_2))
(iso t_2 t_3 (R_1 R_4) (R_2 R_3))))

The first sub-list contains the contacts and terminals, the second is the graph structure of the circuit (every single node is followed by the list of direct neighbors), and the third list is the set T of measurements (conduction or isolation between two terminals, *e.g.* t_3 and t_4, with a certain state, *i.e.,* a set of excited and a set of released relays).

In the following, C will refer to the circuit description (the first two sublists), and T will refer to the third sublist, *i.e.,* the test-plan. C represents the input of the Relais Master, and T its output. C and T together form the input of the correctness predicate $check_Q$.

2.2 The Correctness Requirement

For simplicity, we just consider relay groups that contain terminals, connectors (wires, soldering points etc.) and relay contacts as *elements.* Diodes, resistors and capacitors are left out. The generated test-plan consists of *conduction* and *isolation* tests. Note however, that relay contacts are part of relays, so can not be switched independently, and the test-roboter can only measure between terminals (outside connectors). Since later the certification is supposed to be performed automatically, the generated test-plan and hence the Relais Master is *safety critical.* We need the following guarantee:

> *Every* defect of any relay contact, connector or terminal of the physical device will be detected by at least one measurement of the test-plan.

The combination of the generated tests must assure, if successful, that every single element is individually tested to work properly, *i.e.,* to conduct (terminals, connectors and relay contacts) and isolate (relay contacts) if supposed to. In that case, we call the test-plan T *complete for* the circuit C. The crucial point is, that this is a *partial correctness* requirement for the Relais Master program (`R` for short): If it successfully generates a test-plan T for a given circuit C, then we want T to be *correct.* Let $P(C)$ characterize regular circuit descriptions, and let $Q(T)$ define T to be *complete for* C. Then we may formalize the correctness requirement for `R` by

$$\{\, P(C) \,\}\ T \mathtt{:=R}\,\mathtt{(}C\mathtt{)}\ \{\, Q(T) \,\}\,.$$

Partial correctness requirements are typical for tools used in the construction of safety critical applications. We do not want to prove that the tool never fails. But we want to guarantee that any given result is correct, *i.e.,* that the tool is partially correct w.r.t. the correctness requirement for its results.

2.3 Runtime Verification of Results - Getting the Idea

Let us assume a transformational program `y :=` π`(x)` (or π for short) to be specified by pre- and post-conditions $P(x)$ and $Q(y)$ for inputs x and outputs y. Instead of proving the partial correctness (cf. [19, 22])

$$\{\, P(x) \,\}\ y \mathtt{:=} \pi\ \mathtt{(}x\mathtt{)}\ \{\, Q(y) \,\}$$

of the program π itself – *i.e.,* if P holds for x, and if π successfully terminates on x with result y, then Q holds for y – the idea is to modify the program and add a *checker predicate* which rejects incorrect results. That is to say, we construct a sufficient algorithmic formulation check_Q of the post-condition Q and prove

$$\mathit{check}_Q\,(x,y) \,=\, \mathit{true} \implies Q\,(y)\,.$$

We call this property the *checker correctness.* It is now a simple exercise to prove that the modified program π', which additionally checks the post-condition and rejects any incorrect result, is partially correct with respect to P and Q:

$$\{\, P(x) \,\}\ y := \pi\,(x)\,;\ \mathbf{if}\ \neg\, \mathit{check}_Q\,(x,y)\ \mathbf{then\ abort\ fi}\ \{\, Q(y) \,\}$$

Of course we additionally have to guarantee, that running π does not corrupt check_Q. In our case, we run a separate checker program on ASCII representations of x and y, not modifying the expert system at all.

2.4 The Checker and its Verification

In order to exploit the runtime result verification approach we just have to construct (and to verify) a predicate $check_Q$ on circuits C and test-plans T which checks T to be complete for C, *i.e.,* which guarantees that the postcondition $Q(T)$ holds. Thus, in order to verify the Relais Master expert system it remains to prove such a $check_Q$ to be *sufficient* to guarantee Q, *i.e.,*

$$check_Q\,(C,T) = true \implies Q\,(T)\,.$$

We prove that the checker program returns *true* only if the measures in T are sufficient to guarantee that every contact c_i in C is individually tested both to conduct, if it should be *closed*, and not to conduct, if it should be *open*. Terminals and single connections of topological connectors can be seen as special *closed* contacts. Note, that for safety it is again sufficient to prove *partial correctness* of the checker, *i.e.,* the checker might fail even if Q holds. We have to prove, that the test-plan T is complete for the circuit C, if the checker returns *true* for C and T.

We cannot go into much detail of the program or the proof. The checker is a Lisp program. Inputs are s-expressions representing C and T (as in the above example). C is a graph with nodes for every element, T is a set of either conduction or isolation tests between two terminals in a given state of the relay contacts.

A successful conduction test between t_1 and t_2 guarantees *one* conducting path (we do not know which), whereas a successful isolation test guarantees *every* path to isolate. In both cases, however, this might well be due to a defect. The checker proceeds in five steps:

1. The circuit representation C is transformed into a handier internal graph representation.
2. Each test in T is replaced by the set of all paths between t_1 and t_2 in C together with its type.
3. Every contact of each path in a test is augmented with the corresponding should-be-state (open or closed).
4. Assuming all tests to succeed, we get true logical propositions about physical conduction of (sets of) paths in the concrete device; simple logical transformations give us true propositions about individually tested single elements.
5. Finally, we check that indeed every circuit element is individually tested. If not, the checker aborts.

The checker is written in the subset of applicative Common Lisp supported by the Boyer/Moore theorem prover ACL2 [24]. More details on the checker and on the correctness proof can be found in [3, 2]. The crucial part (step 4) can be seen as a problem specific tautology or model checking [7, 23]: Successful hardware-in-the-loop test according to the generated set T of measurements will guarantee a circuit dependent set of logical formulas expressing conduction or isolation of paths between terminal contacts to be true, and our checker program basically checks the particular formula

$$\bigwedge_{l=1}^{p} t_l \ \wedge \ \bigwedge_{j=1}^{k} tc_j \ \wedge \ \bigwedge_{i=1}^{n} (c_i^{closed} \wedge \neg\, c_i^{open})$$

(for all p terminals, k topological connectors and all n relay contacts) to be a logical consequence of that set of formulas: Every terminal, topological connector and relay contact conducts if closed, and every relay contact isolates if open. If so, the set T is *complete* for the given circuit, *i.e.,* the (finite) set T of *test cases* is sufficient for certification of the devices.

Runtime result verification [13, 35] is strongly related to program checking [4], and our case study is an application in the field of testing safety critical devices. However, our main focus is on verification of the checker program that guarantees correctness of the test case generator (the Relais Master expert system). In contrast to *e.g.* [20], where test case generation is based on model checking, our approach does not rely on particular techniques used in the expert system itself. Classical verification and also model checking might well find their limits for complex (and large) applications. We strongly believe that (verified) runtime result verification scales up in certain mission critical domains – not only for safety but also for security. The approach compares to Necula's proof-carrying code and certifying compilers [32, 33], but we check *sufficient* and not only necessary conditions for correctness.

Compared to the Relais Master expert system, the checker is a recursive correctness predicate of moderate size written in ACL2 Lisp. Its (partial) correctness is proved and the proof is mechanically checked by ACL2. However, in order to guarantee that we can trust test plans generated by execution of the checked Relais Master, we have to prove safety, *i.e.,* partial correctness, for the checker executable as well. And this brings us to the second running example: compiler verification.

3 Trusted Execution of Programs

Compilers are sequential transformational programs. A compiler takes a (syntactical representation of) a *source program* $\pi \in \mathbf{SL}$ as input and sometimes it successfully terminates and returns a *target program* $m \in \mathbf{TL}$. If so, we hope that m has something to do with π. Actually, we want a guarantee that m is a *correct implementation* of π. But this needs further explanation and we will go into some detail on the precise meaning of correct implementation later (section 4). A practical compiler, however, will fail in most cases. Actually, it will fail on nearly every source program (with the precise mathematical meaning of *nearly every*). There are usually infinitely many and arbitrarily large source programs, whereas the compiler will run on a finite machine with hard resource restrictions.

Thus, with trusted execution on a real machine in mind, we can not hope to be able to prove that a compiler (executable) will succeed on every proper source program. But we can prove that if it succeeds and returns a machine program m, then m is a correct implementation of π. The intuition behind this kind of requirements is crucial to many other transformational programs as well, like proof checkers, model or equivalence checkers, batch runs that move our money from one to another account and so on. We accept failures, but we do not want to see incorrect non-erroneous results [36]. This is in fact what we often depend on. Not more. In our every day experience using programs we observe them to fail with a segmentation fault or bus error, for instance due to lack of memory, a programmer's error, a compiler bug, or a misuse of an optimizing compiler under wrong assumptions. Although very annoying, we all live with software errors, but we all hope that the application programmers, compiler constructors, operating system designers and even hardware engineers have been sensible enough to detect and signal any such runtime error. Undetected errors might have harmful consequences, in particular if they are intentional, in which case we would call them a *virus* or *Trojan Horse* [9].

As a compiler constructor, we can not relieve the application programmer from the burden to prove applications correct. We cannot guarantee correctness of implementations for incorrect source programs. Actually we have to construct the compiler without any knowledge about the intended meaning of source programs. On the other hand, a compiler can not preserve every property of the source program. We have to negotiate on a contract between user, language designer and implementor that involves the concrete description of the concrete language to be implemented, the concrete description of the target machine, definitions of both which are sufficiently mathematically exact so that user and implementor can agree on them without misunderstanding. Once such a contract has been negotiated, however, all bets are off for properties of programs not mentioned in that contract.

Some Requirements

Let us collect some requirements for a realistic notion of correct implementation: It should

- handle resource limitations of the target machine,
- handle non-determinism of the source program semantics,
- allow for optimizations
- allow for trusted execution also of non-terminating reactive programs,
- and support full recursion and dynamic data types of source languages.

We often think of source programs to serve as specifications of their machine implementations and thus require that the target program should at least be as defined as the source program and hence be able to compute every source program result. That is to say, we require the classical implementation correctness notion of specification refinement. On the other hand, we often do not want to see incorrect results computed by target program executions. That is to say every target program result should be correct with respect to the source program semantics. If we summarize these two requirements, the target program should

– at least be able to compute every result which the source program specifies
– at most compute correct results with respect to the source program semantics

The careful reader will already have realized, that in general these two requirements contradict. They only meet in the rare cases where we can guarantee the target program to be defined if and only if the source program is, and in case of non-deterministic source program semantics we even would guarantee the target program to be able to compute every source program result. Implementations are often more deterministic and less defined than source programs. It depends on the application domain of user programs if we prefer a notion of correct implementation which corresponds to the first or to the second intuitive requirement.

For the every day programming situation a requirement corresponding to the second item above is the adequate correctness notion. It does not require well-definedness proofs for source programs and guarantees a well-placed trust in target computations even if source programs have not been verified.

> Thesis: We can *trust machine program execution* if we can guarantee partial correctness [19] for the machine program with sufficient mathematical rigor.

Note that in this context *to be undefined* means not to return any non-erroneous result at all, *i.e.,* to abort with a runtime error or not to terminate. Our intuitive compiler correctness requirement is to guarantee partial correctness of the generated machine program. But we did not yet say what kind of correctness property we expect for the source program in order to be able to give such a guarantee. This question actually opens up a deep and subtle discussion on source program properties which we want to preserve or need for instance for optimization (see [31, 17]). As long as we do not really depend on sophisticated optimizations, or on a termination guarantee for the target program, partial source program correctness is sufficient, which means that we require a correct compiler to *preserve partial program correctness* [12, 30].

There is a subtle difference to specification refinement (like in VDM [22]) or to former work on compiler verification using ACL2 resp. its predecessor Nqthm ([27, 8, 28]). J Moore proves in [28], that every non-erroneous result of (the Piton machine on) π will also be computed by m (on the FM9001), that m is more defined than π. This allows for optimizations, but trusted execution of m requires total correctness of π.

4 Compiler Correctness for Transformational Programs

Let us model the semantics of transformational programs by partial relations (multivalued partial functions) $f \subseteq D_i \times D_o$ between input domains D_i and output domains D_o. Thus f is an element $f \in (D_i \rightharpoonup D_o) =_{\text{def}} \mathbf{2}^{D_i \times D_o}$. Data in D_i and D_o are considered *regular* or *non-erroneous* input/output data or states. In order to handle irregular data, *i.e.,* finite and infinite errors, we assume that every data domain D is *extended* by an individually associated disjoint non-empty error set Ω, *i.e.,* $D^{\Omega} =_{\text{def}} D \dot{\cup} \Omega$ and $D \cap \Omega = \emptyset$. For every transformational program semantics f we assume an *extended* version

$$f \in (D_i{}^{\Omega} \rightharpoonup D_o{}^{\Omega})$$

which we denote with the same symbol f. Errors are final. No computation will ever recover from an error [2]. Thus, we require (the extended) f to be *error strict, i.e.,* f to be total on Ω_i and $f(\Omega_i) \subseteq \Omega_o$.

However, we have to respect a further phenomenon. Errors are of essentially different types. They are either unavoidable and we have to accept them, or they are unacceptable and thus to be avoided. The implementation has to guarantee that acceptable errors are signaled and hence do not lead to unexpected incorrect results, whereas the absence of unacceptable errors has to proved by the user, for instance if she/he wants to use an optimizing compiler which omits array boundary checks. We will allow unacceptable errors to cause unpredictable (or chaotic) consequences. In order to model this phenomenon, we partition Ω in a non-empty set $A \subseteq \Omega$ of *acceptable* and a non-empty set $U =_{\text{def}} \Omega \setminus A$ of *unacceptable* or *chaotic* errors. So we require

$$\Omega_i = A_i \dot{\cup} U_i \quad \text{and} \quad \Omega_o = A_o \dot{\cup} U_o$$

and a *strong* error strictness of f, namely that f is total on Ω_i and

$$f(A_i) \subseteq A_o \quad \text{and} \quad f(U_i) \subseteq U_o\,.$$

The error sets Ω are supposed to contain a particular standard error element $\perp$ which is to model infinite computation (divergence). We additionally require $(d, \perp_o) \in f$ whenever there is a non-terminating (infinite) computation of f starting with $d \in D_i{}^{\Omega}$.

Correct Implementation

Let f_s be source and f_t target program semantics, and let $\rho_i \in (D_i^{s\,\Omega} \rightharpoonup D_i^{t\,\Omega})$ and $\rho_o \in (D_o^{s\,\Omega} \rightharpoonup D_o^{t\,\Omega})$ be data representation relations between source and target input and output data. We require both relations, ρ_i and ρ_o, and their inverses $\rho_i{}^{-1}$ and $\rho_o{}^{-1}$, to be strongly error strict (in both directions).

Definition 1 (Correct implementation or refinement). *f_t is said to be a* correct implementation *or* refinement *of f_s* relative *ρ_i and ρ_o and associated error sets iff*

$$(\rho_i\,;\,f_t)(d) \subseteq (f_s\,;\,\rho_o)(d) \cup A_o^t \tag{1}$$

holds for all $d \in D_i^s$ with $f_s(d) \subseteq D_o^s \cup A_o^s$ (that is with $f_s(d) \cap U_o^s = \emptyset$).

[2] Exceptions are not errors in our sense. We think of exceptions as special cases of non-local regular control flow.

$$\begin{tikzcd} D_i^{s\,\Omega} \arrow[r, "f_s"] \arrow[d, "\rho_i"'] & D_o^{s\,\Omega} \arrow[d, "\rho_o"] \\ D_i^{t\,\Omega} \arrow[r, "f_t"'] & D_o^{t\,\Omega} \end{tikzcd}$$

Fig. 2. Source and target program semantics f_s, f_t and data representations ρ_i, ρ_o

For any target program computation, the outcome d'' in $D_o^{t\,\Omega}$ is either an acceptable error output in A_o^t, or there exists a source program computation that either ends in a (regular) d' corresponding to d'' or that ends with an unacceptable (chaotic) error output in U_o^s. That is to say: If f_t is a correct implementation of f_s, then f_t either returns a correct result, or an acceptable error, or, if f_s can compute an unacceptable error, f_t may (chaotically) return any result.

If f_t correctly implements f_s relative ρ_i and ρ_o, we will write $\rho_i \,;\, f_t \sqsupseteq f_s \,;\, \rho_o$ or even shorter just $f_t \sqsupseteq f_s$ (with a boldface $\sqsupseteq$). We indicate this diagram commutativity by the diagram in Figure 3 and we can easily prove that we

$$\begin{tikzcd} f_s \in \mathrm{Sem}_s \arrow[d, "\sqcap\!|"'] & : & D_i^{s\,\Omega} \arrow[r, "f_s"] \arrow[d, "\rho_i"'] \arrow[dr, phantom, "\sqsupseteq"] & D_o^{s\,\Omega} \arrow[d, "\rho_o"] \\ f_t \in \mathrm{Sem}_t & : & D_i^{t\,\Omega} \arrow[r, "f_t"'] & D_o^{t\,\Omega} \end{tikzcd}$$

Fig. 3. Commutative diagram expressing correct implementation

can compose correct implementation diagrams both vertically and horizontally. Correct implementation is compositional, which is a very important fact for practical software engineering and in particular for compiler construction and bootstrapping (see [17] for details).

It is an important observation, that we can exactly characterize correct implementation by *preservation of relative correctness* [37], which generalizes Floyd's and Hoare's notions of partial respectively total program correctness. Let $f \in (\, D_i^{\,\Omega} \rightharpoonup D_o^{\,\Omega}\,)$ be a program semantics and let $P \subseteq D_i$ and $Q \subseteq D_o$ be predicates, *i.e.,* subsets of regular data. f is called *(relatively) correct with respect to* (pre- and post-conditions) P and Q ($\langle\, P \,\rangle\, f \,\langle\, Q \,\rangle$ for short), iff

$$f(P) \;\subseteq\; Q \,\cup\, A_o \tag{2}$$

The following theorem says that f_t correctly implements f_s if and only if relative correctness of f_s implies relative correctness of f_t for all pre- and post-conditions P and Q. A proof can be found in [17], where we also discuss some variations of the notions defined here.

Theorem 1 (Preservation of relative correctness). *f_t correctly implements f_s ($\rho_i; f_t \sqsupseteq f_s; \rho_o$) if and only if for all $P \subseteq D_i^s$ and $Q \subseteq D_o^s$ we have*

$$\langle P \rangle f_s \langle Q \rangle \implies \langle \rho_i(P) \rangle f_t \langle \rho_o(Q) \rangle .$$

Relative program correctness generalizes the classical notions of partial or total program correctness. In order to see this, let $f \in (D_i \rightharpoonup D_o)$ be an original unextended program semantics and let $P \subseteq D_i$ and $Q \subseteq D_o$ be pre- and postconditions. f is called *partially correct w.r.t.* P and Q ($\{P\} f \{Q\}$ for short), if $f(P) \subseteq Q$, *i.e.,* if the following holds for f: Whenever the precondition P holds for an input $d \in D_i$ and if $f(d) \subseteq D_o$ is defined, then any $d' \in f(d)$ fulfills the postcondition Q. And f is called *totally correct w.r.t.* P and Q ($[P] f [Q]$ for short), if f additionally is guaranteed to be defined on any $d \in P$, *i.e.,* if f is partially correct w.r.t. P and Q and the domain dom_f of f comprises P (that is if additionally $\mathrm{dom}_f \supseteq P$ holds).

Let us choose particular error sets $A = A_i = A_o =_{\mathrm{def}} \{a\}$ and $U = U_i = U_o =_{\mathrm{def}} \{u\}$ with acceptable error element a and unacceptable error element u for both domains D_i and D_o. Hence, $\Omega = \Omega_i = \Omega_o =_{\mathrm{def}} A \stackrel{\bullet}{\cup} U$. Note that $\perp \in \{a, u\}$. We may now consider $\perp$ acceptable ($A_i = A_o =_{\mathrm{def}} \{\perp\}$, $U_i = U_o =_{\mathrm{def}} \{u\}$) or unacceptable ($A_i = A_o =_{\mathrm{def}} \{a\}$, $U_i = U_o =_{\mathrm{def}} \{\perp\}$). Relative correctness of the extended f^{ext} is equivalent to partial correctness of f

$$\langle P \rangle f^{ext} \langle Q \rangle \iff \{P\} f \{Q\}$$

if $\perp = a$ is considered acceptable, and it is equivalent to total correctness, if $\perp = u$ is considered unacceptable:

$$\langle P \rangle f^{ext} \langle Q \rangle \iff [P] f [Q] .$$

But our notion of correct (relative) implementation also generalizes well-known implementation correctness notions. If we consider $\perp$ acceptable, then

$$\rho_i^{ext}; f_t^{ext} \sqsupseteq f_s^{ext}; \rho_o^{ext} \iff \rho_i; f_t \subseteq f_s; \rho_o$$

exactly expresses preservation of partial program correctness, and if we consider $\perp$ unacceptable, on the other hand, then preservation of relative correctness is equivalent to preservation of total correctness (see again [17] for details).

In particular, if we come back to our earlier remarks, preservation of partial program correctness often suffices if we want to trust in target program execution and if we do not depend on sophisticated optimizations. It gives us a mathematically rigorous guarantee that target programs compute *at most* correct results. Hence, preservation of partial correctness ([12, 30, 29], or L-simulation [5]) is a practically motivated adequate special case:

Definition 2 (L-simulation). *Let π and m denote source and target programs with semantics $[\![\pi]\!]_{\mathbf{SL}} \in (D_i^{\mathbf{SL}} \rightharpoonup D_o^{\mathbf{SL}})$ and $[\![m]\!]_{\mathbf{TL}} \in (D_i^{\mathbf{TL}} \rightharpoonup D_o^{\mathbf{TL}})$, respectively. Then we say that the implementation step $\pi \mapsto m$* preserves partial correctness *(or that m* L-simulates *π), if and only if*

$$\rho_i ; [\![m]\!]_{\mathbf{TL}} \subseteq [\![\pi]\!]_{\mathbf{SL}} ; \rho_o . \tag{3}$$

That is to say: If the target program m is defined on the representation of a regular source program input, then the source program π can return a corresponding result as well. If π and hence m are deterministic, this means that π will return *the* corresponding result.

$$\begin{tikzcd}
{[\![\pi]\!]_{\mathbf{SL}} \in \mathrm{Sem}_{\mathbf{SL}}} \arrow[d, phantom, "\bigcup\!|"] \arrow[d] & : & D_i^s \arrow[rr, "{[\![\pi]\!]_{\mathbf{SL}}}"] \arrow[d, "\rho_i"'] \arrow[drr, phantom, "\circlearrowleft"] & & D_o^s \arrow[d, "\rho_o"] \\
{[\![m]\!]_{\mathbf{TL}} \in \mathrm{Sem}_{\mathbf{TL}}} & : & D_i^t \arrow[rr, "{[\![m]\!]_{\mathbf{TL}}}"'] & & D_o^t
\end{tikzcd}$$

Fig. 4. Commutative diagram expressing preservation of partial program correctness

Corollary 1 (Preservation of partial correctness). *An implementation step $\pi \mapsto m$ preserves partial correctness if and only if m L-simulates π, i.e.,*

$$\begin{aligned} &\forall\, P, Q : (\, \{\, P \,\}\, \pi \,\{\, Q \,\} \implies \{\, \rho_i(P) \,\}\, m \,\{\, \rho_o(Q) \,\}\,) \\ &\iff \rho_i \,;\, [\![\, m \,]\!]_{\mathbf{TL}} \subseteq [\![\, \pi \,]\!]_{\mathbf{SL}} \,;\, \rho_o \end{aligned}$$

Preservation of relative correctness gives us the necessary means to define adequate and more elaborate notions of correct implementation also for realistic correct compilation, not only of preservation of partial correctness, but also for optimizing compilers where target program inputs have to fulfill additional optimization (total correctness) conditions (do not cause particular runtime errors) for the target program to run trustworthily [17].

We may compare the previous section to a *requirements analysis* as of usual software engineering processes. We might even call it a requirements analysis in proof engineering for the correctness of compilers. The result is a mathematical framework which allows us to precisely and also formally express the theorem(s) which we have to prove in order to call a compiler correct.

5 Proving Compiler Source Level Correctness

Let us now give an example and sketch through a concrete compiler source level correctness proof which we mechanically ran through the Boyer/Moore theorem prover ACL2 [24].

In [9, 10] we prove preservation of partial correctness for a Lisp compiler, and we want to give a sketch and some brief remarks on source and target language, the compiler and its correctness proof. In [9] we define an abstract stack machine, and a compiler that generates stack machine code for a small subset of ACL2. The compiler is written in its own source language so that we can execute it within ACL2, either directly as an ACL2 function, or after compilation (into abstract machine code) by executing the machine model and running the compiled compiler (machine) program on it. The latter simulates machine execution of compiler executables within ACL2, and the main focus of [9] has been on that kind of (machine) program execution. We actually proved formally and rigorously that source level verification is not sufficient to finally guarantee correctness and hence trustworthiness of compiler executables. The compiler correctness proof is part of the most recent ACL2 distributions and described in detail in [10].

The source language **SL** is a subset of ACL2 Lisp similar to the language L3 of first order mutually recursive functions as of [26], but with s-expressions $s \in \mathsf{SExpr}$ as (dynamic) data. A program is a list `defs` of function definitions, followed by a list `vars` of input variables and a *main* program expression. The (operational) semantics $[\![\,\pi\,]\!]_{\mathtt{SL}} \in (\,\mathsf{SExpr}^* \;\rightarrow\; \mathsf{SExpr}\,)$ is defined by an ACL2 function `evaluate` which (deterministically) maps `input` values to the result of the `main` expression after binding the inputs to the input variables. A termination argument, a natural number `n`, has been added in order to model strict partial functions within the ACL2 logic of total recursive functions (see [10] for more details).

The target language **TL** is the code of an an abstract stack machine. Its configuration consists of a `code` part and a separate state `stack`, which is a data stack containing Lisp s-expressions. The machine has six machine instructions including a subroutine call and a structured conditional. Machine programs (m) are sequences of (mutually recursive) subroutine declarations (d) together with a *main* instruction sequence which is to be executed on an initial `stack` after downloading the list of declarations into `code`. The semantics $[\![\,\mathtt{prog}\,]\!]_{\mathtt{TL}} \in (\,\mathsf{SExpr}^* \;\rightarrow\; \mathsf{SExpr}^*\,)$ of machine programs is a `stack`-transformation defined by stepwise executing machine instructions. It is defined by an ACL2 function `execute` which executes a program `prog` on an initial `stack` which is supposed to contain input values on top. Again, a termination argument `n` is added in order to force the machine to stop execution after at most `n` subroutine calls.

The compiler `compile-program` is written in **SL** itself. It generates stack machine code according to the *stack principle*, *i.e.,* arguments are passed on the stack and a given expression e is compiled to a sequence of machine instructions pushing the value of e onto the stack leaving the stack below invariant. Functions or operators consume (pop) their arguments and push the result.

We prove that `compile-program` preserves partial correctness (Figure 5). The proof is mechanically checked by ACL2 and the complete ACL2 proof script is part of the recent ACL2 distribution. For those readers familiar with ACL2 let us cite the main theorem from [10]:

```
(defthm compiler-correctness-for-programs
 (let ((new-stack (execute (compile-program prog)
                           (append (rev inputs) old-stack) n))
       (value (car (evaluate prog inputs n))))
  (implies (and (wellformed-program prog)
                (defined new-stack)
                (true-listp inputs) (equal (len vars) (len inputs)))
           (equal new-stack (cons value old-stack)))))
```

$$\begin{tikzcd}
(s_1,\dots,s_n) \in \mathsf{SExpr}^* \arrow[r, "\texttt{(evaluate } \pi \cdot \texttt{n)}"] \arrow[d, "\rho_{tos}"'] & \mathsf{SExpr} \ni s \arrow[d, "\rho_{tos}"] \\
(s_n,\dots,s_1,\dots) \in \mathsf{Sexpr}^* \arrow[r, "\texttt{(execute } m \cdot \texttt{n)}"'] & \mathsf{Sexpr}^* \ni (s,\dots)
\end{tikzcd}$$

Fig. 5. `compile-program` preserves partial correctness; the data representation relation ρ_{tos} relates s-expressions (or s-expression lists) to target machine `stacks` which contain the s-expression (list) on top (in reverse order)

If the source program is well-formed, and if the target program (applied to an initial stack containing the correct number of `inputs` in reverse order on top) returns a non-erroneous result on top (is `defined`), then this result is `equal` to the semantics (`evaluate`) of the program applied to the inputs.

The crucial part of the proof, however, is to prove correctness of expression compilation. The correctness theorem for programs is, after some technical lemmas, a simple consequence of a similar theorem for correct expression compilation, applied to the main expression of the program. For correct expression compilation we prove two theorems simultaneously by induction. The first is the correctness theorem for forms (or expressions), and the second is the theorem for form lists. Both of them are very similar. Informally, for well-formed expressions and expression lists in well-formed programs we prove: If the machine, executed on a compiled expression (list), is defined on a `stack` for an `n`, then the following three conjectures hold:

1. The semantics of the expression (list) – in the given function environment and with the free variables bound to their values in the current stack-frame – is defined for the same `n`,
2. the machine returns a new stack with the value(s) of the expression(s) on top (in reverse order), and
3. the stack just below the result value(s) remaines unchanged.

The proof is actually a combined computational and structural induction on the termination argument `n` and the structural depth of the expression (in ACL2 terminology this is a well-founded induction on the ordinals). The induction is suggested by a large admissible ACL2 function which explicitly lists the entire set of induction hypotheses that we need for the proof to succeed.

We need a lot of lemmas. Most of them are technical, but some reflect crucial insights necessary to find the proof. So, for instance, it is interesting, that we actually have to prove the definedness of the source code semantics and the correctness of the result of machine execution simultaneously as well. The reason is the conditional. The definedness of the conditional inductively depends on the value of the condition. The conditional needs not be strict in both alternatives, and will not be for instance in recursive definitions. The conditional has actually been the challenging case to find the proof, not the function application as the reader might have expected. Function application is just captured by computational induction, whereas the conditional crucially influences the proof structure in the large. But we do not want to go into further detail. We refer to [9, 10] instead.

6 On the Impact of Toy Examples

In a sense, our example above is a toy example compared to the work necessary for the verification of realistic compilers for practical source programming languages and real target processors [17]. Our source language is very small, and the abstract target stack machine code is unrealistically abstract and far away from the ugly reality of real world processors. In that sense, [10] only presents an exercise. But the proof is an interesting exercise, and it contributes to realistic compiler verification in two important respects:

- It can be accomplished by the mechanical verification of subsequent compilation phases, so for instance (b) to a stack machine with a linear heap store (data refinement), (c) to linear assembly code, and (d) to real binary machine code of a concrete processor [15].
- In the context of the *Verifix* project on correct compilers [12, 15, 18], the proof has been generalized to a mechanical PVS [34] proof of preservation of partial correctness for a larger imperative source language [6].

The correctness of step (b) has already been proved manually, and the mechanical proof is nearly completed [11]. And step (c) and (d) are implemented [15, 16, 14] and designed to preserve partial correctness. Thus, in addition to being a nice exercise, our proof can be seen as part of a medium to large scale proof effort to provide an initial fully verified compiler executable that preserves partial source program correctness [15].

In the PVS proof we use structural operational semantics and inductive relations to formalize partial functions. It is interesting, that we can essentially reuse the proof idea and overall proof structure of the ACL2 proof presented here; The PVS proof also is by combined structural and computational induction. In our opinion, this is a good news with respect to proof engineering.

In both respects we benefit from the toy example. It shows that such a proof is manageable in principle, without premature introduction of realistic complexity. It actually has been the first mechanical proof of preservation of partial correctness. And such a proof is necessary in order to provide a mathematically rigorous proof also of the correctness of binary compiler executables. In fact, the latter has been the original motivation for the Verifix compiler correctness project: If we bootstrap compilers, we need a guarantee of the partial correctness of the generated compiler executable. We need to handle hard resource limitations of target machines and support for full recursion and dynamic data types of source languages.

7 Conclusions

Software verification can succeed, although it is often considered hard or even impossible – in particular for large and complex systems. Mechanical proof support can help finding proofs (and errors). It may help to redo (or reuse) proofs after slight modifications. And it might establish correctness. But in the latter case we depend on the prover. Its positive results, i.e., the successful proofs, should in principle be understandable and/or checkable, manually or by trusted proof checking programs. Otherwise we would leave a serious gap in the mechanically supported process of correct software construction and proof documentation.

Thus, we also have some wishes, in particular with respect to *proof checkability* and to *intuitive modeling*: First of all, in order to convince our customers, it is not only necessary that a mechanical proof finally succeeds. Once it does, the prover should also contribute to a consistent and complete proof documentation. If proofs are not mechanically checkable by trustworthy proof checkers, they should be readable so that we can check them manually. This is very similar to programming: It is at the end not sufficient that a program finally runs. Once it does, we also need documentation. We need to know and to communicate why.

The second wish is a bit more technical. Compiler verification deals with programming language semantics, and very often programs mean partial functions. In ACL2, we modeled them by error strict total functions, which is kind of tedious and always calls for an additional justification. In PVS, we use inductive relations, which are also a bit tedious to handle. In both cases, however, we need meta-arguments to justify the formalization. Although we know about the problems, we sometimes wish to have a logic of partial functions, and a supporting theorem prover which is as usable and elaborate.

Acknowledgments We would like to thank our colleagues in the *Verifix* project for, in particular Axel Dold, Thilo Gaul, Gerhard Goos, Andreas Heberle, Friedrich von Henke, Ulrich Hoffmann, Hans Langmaack, Vincent Vialard, Wolf Zimmermann. Special thanks to Hans Langmaack, and to Markus Müller-Olm and Andreas Wolf for their contributions to the notion of relative program correctness and its preservation. We thank Roy Bartsch for his work and Uwe Haferstroh, Sven Nordhoff and DTK for supporting the work on the Relais Master case study.

References

1. RelaisMaster Documentation and Manuals. DTK - Gesellschaft für technische Kommunikation MBH, Palmaille 82, Hamburg, Germany, 1996.
2. R. Bartsch and W. Goerigk. Mechanical a-posteriori Verification of Results: A Case Study for a Safety Critical AI System (Abstract). In *AAAI Workshop on Model Based Validation of Intelligence MBVI'2001*, Stanford, CA, U.S.A., March 2001.
3. Roy Bartsch. Mechanisch verifizierte Programmprüfung für die Korrektheit von Prüfplänen in der Bahntechnik. Master's thesis, Institut für Informatik, CAU, Kiel, 2000.

4. M. Blum, M. Luby, and R. Rubinfeld. Program result checking against adaptive programs and in cryptographic settings. In *DIMACS Workshop on Distributed Computing and Crypthography*, 1989.
5. W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*, volume 47 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, New York, NY, 1998.
6. Axel Dold and Wolfgang Goerigk. Proving Preservation of Partial Correctness in PVS. Technical report, University of Ulm, 2000. In preparation.
7. E.M. Clarke and E.A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In D. Kozen, editor, *Proceedings of the Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71, Yorktown Heights, New York, May 1981. Springer-Verlag.
8. A. D. Flatau. *A verified implementation of an applicative language with dynamic storage allocation.* PhD thesis, University of Texas at Austin, 1992.
9. W. Goerigk. Compiler Verification Revisited. In M. Kaufmann, P. Manolios, and J S. Moore, editors, *Computer Aided Reasoning: ACL2 Case Studies.* Kluwer Academic Publishers, 2000.
10. W. Goerigk. Proving Preservation of Partial Correctness with ACL2: A Mechanical Compiler Source Level Correctness Proof. In M. Kaufmann and J S. Moore, editors, *Proceeding of the ACL2'2000 Workshop*, University of Texas, Austin, Texas, U.S.A., October 2000.
11. W. Goerigk. Trusted Program Execution. Habilitation thesis. Techn. Faculty, Christian-Albrechts-Universität zu Kiel, May 2000. To be published.
12. W. Goerigk, A. Dold, Th. Gaul, G. Goos, A. Heberle, F.-W. von Henke, U. Hoffmann, H. Langmaack, H. Pfeifer, H. Ruess, and W. Zimmermann. Compiler Correctness and Implementation Verification: The *Verifix* Approach. In P. Fritzson, editor, *Proceedings of the Poster Session of CC '96 – International Conference on Compiler Construction*, pages 65 – 73, IDA Technical Report LiTH-IDA-R-96-12, Linkøping, Sweden, 1996.
13. W. Goerigk, Th. Gaul, and W. Zimmermann. Correct Programs without Proof? On Checker-Based Program Verification. In *Proceedings ATOOLS'98 Workshop on "Tool Support for System Specification, Development, and Verification"*, Advances in Computing Science, Malente, 1998. Springer Verlag.
14. W. Goerigk and U. Hoffmann. Compiling ComLisp to Executable Machine Code: Compiler Construction. Technical Report Nr. 9812, Institut für Informatik, CAU, October 1998.
15. W. Goerigk and U. Hoffmann. Rigorous Compiler Implementation Correctness: How to Prove the Real Thing Correct. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Applied Formal Methods - FM-Trends 98*, volume 1641 of *Lecture Notes in Computer Science*, pages 122 – 136, 1998.
16. W. Goerigk and U. Hoffmann. The Compiling Specification from ComLisp to Executable Machine Code. Technical Report Nr. 9713, Institut für Informatik, CAU, Kiel, December 1998.
17. W. Goerigk and H. Langmaack. Will Informatics be able to Justify the Construction of Large Computer Based Systems? Technical Report 2015, Institut für Informatik, CAU, 2001.
18. G. Goos and W. Zimmermann. Verification of Compilers. In E.-R. Olderog and B. Steffen, editors, *Correct System Design*, volume 1710 of *LNCS*, pages 201 – 230. Springer Verlag, 1999.
19. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–583, 1969.

20. T. Jeron and P. Morel. Test generation derived from model-checking. *Lecture Notes in Computer Science*, 1633:108–??, 1999.
21. C. Jones, R.E. Bloomfield, P.K.D. Froome, and P.G. Bishop. Methods for assessing the safety integrity of safety-related software of uncertain pedigree (SOUP). Contract Research Report 337/2001, Health and Safety Executive, Adelard, London, UK, 2001.
22. C. B. Jones. *Systematic Software Development Using VDM, 2nd ed.* Prentice Hall, New York, London, 1990.
23. J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the Fifth International Symposium in Programming*, 1981.
24. M. Kaufmann and J S. Moore. Design Goals of ACL2. Technical Report 101, Computational Logic, Inc., August 1994.
25. H. Lange, R. Möller, and B. Neumann. Avoiding Combinatorial Explosion in Automatic Test Generation: Reasoning about Measurements is the Key. In *Proceedings of KI'96 Conference on Artificial Intelligence*, Dresden, 1996. Springer Verlag.
26. Jacques Loeckx and Kurt Sieber. *The Foundations of Program Verification (Second edition).* John Wiley and Sons, New York, N.Y., 1987.
27. J S. Moore. Piton: A verified assembly level language. Technical Report 22, Comp. Logic Inc, Austin, Texas, 1988.
28. J S. Moore. *Piton: A Mechanically Verified Assembly-Level Language.* Kluwer Academic Press, Dordrecht, The Netherlands, 1996.
29. M. Müller-Olm. Three Views on Preservation of Partial Correctness. Technical Report Verifix/CAU/5.1, CAU Kiel, October 1996.
30. M. Müller-Olm. *Modular Compiler Verification: A Refinement-Algebraic Approach Advocating Stepwise Abstraction*, volume 1283 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg, New York, 1997.
31. M. Müller-Olm and A. Wolf. On Excusable and Inexcusable Failures: Towards an Adequate Notion of Translation Correctness. In J. M. Wing, J. Woodcock, and J. Davies, editors, *Proceedings of Formal Methods FM'99*, volume 1709 of *LNCS*, pages 1107–1127, Toulouse, France, 1999. Springer.
32. G. C. Necula. Proof-carrying code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, 15–17 January 1997.
33. George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pages 333–344, Montreal, Canada, 17–19 June 1998.
34. S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, *Proceedings 11th International Conference on Automated Deduction CADE*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, October 1992. Springer-Verlag.
35. A. Pnueli and P. Traverso, editors. *Proceedings of the FLoC'99 International Workshop on "Runtime Result Verification"*, Trento, Italy, 1999.
36. W. Polak. Compiler specification and verification. In J. Hartmanis G. Goos, editor, *Lecture Notes in Computer Science*, number 124 in LNCS. Springer-Verlag, 1981.
37. A. Wolf. *Weakest Relative Precondition Semantics - Balancing Approved Theory and Realistic Translation Verification.* PhD thesis, Technische Fakultät der Christian-Albrechts-Universität, Kiel, 2001.