*Electronic Notes in Theoretical Computer Science 58 No. 2 (2001)*

*URL:* [*http://www.elsevier.nl/locate/entcs/volume58.html*](http://www.elsevier.nl/locate/entcs/volume58.html) *21 pages*

*Mechanical Software Veri cation: High Level* Control Aspects from a User's Perspective

# *Wolfgang Goerigk* [*1*](#_bookmark0) *;*[*2*](#_bookmark0)

*Institut fur Informatik und Praktische Mathematik* *Christian-Albrechts-Universitat*

*Kiel, Germany*

*Abstract*

*We present lessons learned from using mechanical theorem proving for proof support in software veri cation, with trusted execution of programs in mind. We will use two realistic running examples, compiler veri cation, 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 (veri ed) runtime result veri cation. 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 veri cation, i.e., about proving properties of pro-* grams 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 ma- chine 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 de nitely 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 veri cation of programs and compiler veri cation.

*1 Email:* [*wg@informatik.uni-kiel.de*](mailto:wg@informatik.uni-kiel.de)

*2 The work reported here has been supported by the Deutsche Forschungsgemeinschaft (DFG) in the Veri x and VerComp projects on Correct Compilers and Techniques for Compiler Implementation Veri cation*

*c 2001 Published by Elsevier Science B. V. Open access under* [*CC BY-NC-ND license*](http://creativecommons.org/licenses/by-nc-nd/3.0/)*.*

*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 (cf. [[19,22](#_bookmark33)]) and its preservation, i.e., of a correctness guarantee for regularly computed non-erroneous results also of machine executables. This is often suÆcient, 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. 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 nally 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 cir- culi vitiosi, i.e., never ending dependencies on unsafe executable software (of uncertain pedigree, [[21](#_bookmark32)]), we need for principle reasons some manual proof checking. Fortunately, we can push this e ort entirely into compiler imple- mentation correctness proofs and rigorously verify compiler executables to preserve adequate and reasonable correctness properties of source programs [[17](#_bookmark28)].

*We want to demonstrate the proposed modularization in source level and* compiler correctness. For that we use two examples which t together. In section [2](#_bookmark1) we sketch an industrial project on the veri cation of a safety-critical expert system. Its correctness proof has actually been checked by the ACL2 theorem prover [[24].](#_bookmark35) Safety can be proved by a technique which we call (veri-

*ed) runtime result veri cation* [*[13].*](#_bookmark24) *In section* [*3*](#_bookmark3) *we informally collect requi-*

*rements for realistic correct compilation. Section* [*4*](#_bookmark4) *will yield a mathematical* framework which enables us to precisely de ne compiler correctness by preser- vation of source program correctness. In particular, we will sketch a mecha- nical proof of preservation of partial correctness for a toy compiler in ACL2 (section [5).](#_bookmark9) The role of toy examples is crucial in order to nally succeed, and we will give some comments in section [6](#_bookmark11). We will end up with some lessons learned and some wishes for theorem provers used in software veri cation.

# *2 Veri ed Runtime Result Veri cation*

*The e ort of proving the correctness of large software systems seems often* not justi able. Heuristics and programming tricks are necessary in order to to solve complex problems successfully and eÆciently. Mathematical induction then often fails because the algorithms to be veri ed 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*](#_bookmark24)*] we propose a checker-based approach to software veri cation which* exploits the idea of runtime result checking [[4]](#_bookmark15) for veri cation. It is app- licable if partial correctness of the application suÆces. Partial correctness can be proved by a-posteriori runtime result veri cation. Let us sketch an industrial software veri cation project [[3]](#_bookmark14) in which we exploit the technique of veri ed runtime result veri cation (checker-based program veri cation) in order to prove the correctness (soundness) of a safety-critical expert system. The system is called Relais Master [[1](#_bookmark12)[,25],](#_bookmark36) and it is an expert system for com- puting 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 gua- rantee a suÆciently high level of safety, these devices need regular mainte- nance, and in particular, a re-certi cation of every single physical device is required within regular time intervals.

*2.1 The Relais Master |A Safety Critical AI Tool*

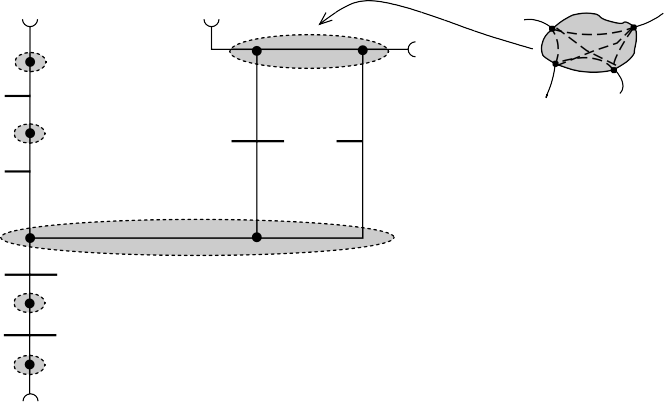
*The Relais Master automates the time-consuming and error prone manual* test-plan construction. The system has been developed by the German com- pany DTK (Hamburg) in cooperation with the Laboratory for Arti cial In- telligence (LKI) at the University of Hamburg, Germany. After scanning the circuit, 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., contain every test necessary to* detect any combination of any number of defects in the device (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 rst 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 so- undness. 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.

*R1 R2*



*t1*

*t3*

*tc*

*c1*

*1*

*t4*

*tc6*

*tc2*

*c*

*R1*

*3*

*R2*

*c2 c4*

*topological*

*connector*

*c5*

*tc3 c6 tc4*

*tc5*

*R3*

*R*

*4*

*t2*

*Fig. 1. A sample circuit. Relais R1 has two contacts, c1 and c2, and R2 has contacts c3 and c4. If R1 is not excited (released), contact c1 is closed (i.e., is to conduct), whereas c2 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.*

*The input of the Relais Master is a circuit description which actually is* manually constructed from a scanned circuit plan. Figure [1](#_bookmark2) shows a very small example. The Ascii-representation of the internal object-oriented Relais Master data structure is a Lisp-s-expression of the following form:

*(((R1 released ((c1 closed) (c2 open)))*

*(R2 released ((c3 closed) (c4 closed)))*

*(R3 released ((c5 open))) (R4 released ((c6 open))) t1 t2 t3 t4)*

*((t1 c1) (t2 c6) (t3 c2 c4 t4) (t4 t3 c2 c4) (c1 t1 c3) (c3 c1 c5 c2 c4)*

*(c5 c6 c3 c2 c4) (c6 c5 t2) (c2 t3 t4 c4 c3 c5) (c4 t3 t4 c2 c3 c5))*

*((con t3 t4 () (R1 R2 R3 R4)) (con t3 t1 () (R1 R2 R3 R4))*

*(iso t2 t1 (R3) (R1 R2 R4)) (con t3 t2 (R3 R4) (R1 R2))*

*(iso t3 t2 (R2 R3 R4) (R1)) (iso t1 t2 (R2 R3 R4) (R1))*

*(con t3 t2 (R1 R2 R3 R4) ()) (iso t1 t2 (R1 R3 R4) (R2))*

*(iso t2 t3 (R1 R4) (R2 R3))))*

*The rst 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, which are* conduction (con) or isolation (iso) tests between two terminals |for instance between t3 and t4 | followed by a certain state given by the set of excited and the set of released relays in that order.

*In the following, C will refer to the circuit description (the rst two sub-* lists), and T will refer to the third sublist, i.e., to the test-plan. C represents the input of the Relais Master, and T represents its output. C and T to- gether form the input for the correctness predicate check Q, which is intended to be implemented by a program which checks suÆcient conditions for T to be complete for C.

*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 certi cation 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 sup- posed 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 ) de ne T to be complete for C. Then we may formalize the correctness requirement for R by the Hoare-triple

*f P (C) g T := R (C) f Q(T ) g :*

*Partial correctness requirements are typical for tools used in the construc-* tion of safety critical applications. We need not 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 pre- and post-conditions expressing the correctness requirement for its results.

*2.3 Runtime Veri cation of Results - Getting the Idea*

*Let us assume a transformational program y := (x) (or for short) to be* speci ed by pre- and post-conditions P (x) and Q(y) for inputs x and outputs

*y. Instead of proving the partial correctness*

*f P (x) g y := (x) f Q(y) g*

*of the program itself |i.e., if P holds for x, and if 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 suÆcient algorithmic formulation check Q of the post-condition Q and prove

*( check Q (x; y) = true ) =) Q (y) :*

*We call this property the checker correctness. It is now a simple exercise* to prove that the modi ed program 0 below, which additionally checks the post-condition and rejects any incorrect result, is partially correct with respect to P and Q:

*f P (x) g y := (x); if : check Q (x; y) then abort f Q(y) g*

*Of course we additionally have to guarantee, that running 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 Veri cation*

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

*( check Q (C; T ) = true ) =) Q (T ) :*

*We prove that the checker program returns true only if the measures in* T are suÆcient to guarantee that every contact ci 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 suÆcient 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 successfully returns true for C and T .

*We cannot go into much detail of the program and 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 t1 and t2 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 ve steps:*

*(i) The circuit representation C is transformed into a handier internal graph* representation.

*(ii) Each test in T is replaced by the set of all paths between t1 and t2 in C* together with its type.

*(iii) Every contact of each path in a test is augmented with the corresponding* should-be-state (open or closed).

*(iv) Assuming all tests to succeed, we get true logical propositions about phy-* sical conduction of (sets of) paths in the concrete device; simple logical transformations give us true propositions about individually tested single elements.

*(v) 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].](#_bookmark35) More details on the checker and on the correctness proof can be found in [[3](#_bookmark14)[,2].](#_bookmark13) The crucial part (step 4) can be seen as a problem speci c tautology or model checking [[7](#_bookmark18)[,23]:](#_bookmark34) 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

*p*

*^ t ^*

*k*

*^ tc*

*n*

*^ ^(cclosed ^: copen )*

*l*

*l=1*

*j*

*j=1*

*i* *i*

*i=1*

*(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 ( nite) set T of test cases is suÆcient for certi cation of the devices.

*Runtime result veri cation* [*[13*](#_bookmark24)[*,35]*](#_bookmark46) *is strongly related to program checking* [[4](#_bookmark15)], and our case study is an application in the eld of testing safety critical devices. However, our main focus is on veri cation of the checker program that guarantees correctness of the test case generator (the Relais Master expert system). In contrast to e.g. [[20](#_bookmark31)], where test case generation is based on model checking, our approach does not rely on particular techniques used in the expert system itself. Classical veri cation and also model checking might well

*nd their limits for complex (and large) applications. We strongly believe*

*that (veri ed) runtime result veri cation 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],](#_bookmark44) but we check suÆcient 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) cor- rectness 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 veri cation.

# *3 Trusted Execution of Programs*

*Compilers are sequential transformational programs. A compiler takes a (syn-* tactical representation of) a source program 2 SL as input and sometimes it successfully terminates and returns a target program m 2 TL. If so, we hope that m has something to do with . Actually, we want a guarantee that m is a correct implementation of . But this needs further explanation and we will go into some detail on the precise meaning of correct implementation later (section [4).](#_bookmark4) A practical compiler, however, will fail in most cases. Actually, it will fail on nearly every source program (with the precise mathematical mea- ning of nearly every). There are usually in nitely many and arbitrarily large source programs, whereas the compiler will run on a nite 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 . 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].](#_bookmark47) 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 appli- cation 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].](#_bookmark20)

*As a compiler constructor, we can not relieve the application programmer* from the burden to prove applications correct. We cannot guarantee correct- ness 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 descrip- tion of the concrete language to be implemented, the concrete description of the target machine, de nitions of both which are suÆciently mathematically exact so that user and implementor can agree on them without misunderstan- ding. Once such a contract has been negotiated, however, all bets are o for properties of programs not mentioned in that contract.

*3.1 Informal Requirements*

*Let us collect some requirements for a realistic notion of correct implementa-* tion: 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 speci cations of their ma-* chine implementations and thus require that the target program should at least be as de ned 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 speci cation re nement like for instance in VDM [[22].](#_bookmark33) 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 [[36].](#_bookmark47) If we summarize these two requirements, the target program should

*at least be able to compute every result which the source program speci es*

*at most compute correct results w.r.t. 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 de ned 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 de ned than source programs. It depends on the application domain of user programs if we prefer a notion of correct implementation which corresponds to the rst or to the second intuitive requirement.

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

*veri ed. For this situation we want to formulate the following* Thesis:

*We can trust machine program execution if we can guarantee partial cor-*

*rectness for the machine program with suÆcient mathematical rigor.*

*Note that to be unde ned here means not to return any non-erroneous* result, i.e., either to abort with a runtime error or not to terminate at all.

*Thus, our intuitive compiler correctness requirement for transformational* programs is to guarantee partial correctness of the generated machine pro- gram. 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](#_bookmark42)[,17]).](#_bookmark28) 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 suÆcient, which means that we require a correct compiler to preserve partial program correctness, which is adequate, useful, and provable also for realistic compiler executables [[15](#_bookmark26)[,17].](#_bookmark28)

*Note that there is a subtle di erence to speci cation re nement or to for-* mer work on compiler veri cation using ACL2 resp. its predecessor Nqthm [([27](#_bookmark38)[,8,28]).](#_bookmark39) J Moore proves in [[28],](#_bookmark39) that every non-erroneous result of (the Piton machine on) will also be computed by m (on the FM9001), that m is more de ned than . This notion corresponds to the rst intuitive requi- rement above, easily allows for optimizations, but trusted execution of the target program m requires total correctness proofs for the source program .

# *4 Compiler Correctness for Transformational Programs*

*Let us model the semantics of transformational programs by partial relations* (multivalued partial functions) f Di Do between input domains Di

*and output domains Do*

*. Thus f is an element f 2 ( Di*

*\* Do*

*) =def*

*2Di Do .*

*Data in Di and Do are considered regular or non-erroneous input/output data*

*or states. In order to handle irregular data, i.e., nite and in nite errors, we*

*assume that every data domain D is extended by an individually associated*

*disjoint non-empty error set , i.e., D =*

*def*

*D [ and D \ = ;. For*

*every transformational program semantics f we assume an extended version*

*f 2 ( D \* D )*

*i o*

*which we denote with the same symbol f . Errors are nal. No computation* will ever recover from an error [3](#_bookmark5) . Thus, we require (the extended) f to be error strict, i.e., f to be total on i and f ( i) o.

*3 Exceptions are not errors in our sense. We think of exceptions as special cases of non-local regular control ow.*

*However, we have to respect a further phenomenon. Errors are of essenti-* ally di erent 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 be proved by the user, for instance if she/he wants to use an optimizing compiler which omits array boundary checks. We will allow unacceptable er- rors to cause unpredictable (or chaotic) consequences. In order to model this phenomenon, we partition in a non-empty set A of acceptable and a non-empty set U =def n A of unacceptable or chaotic errors. So we require

*i = Ai*

*[ Ui and o = Ao*

*[ Uo*

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

*f (Ai) Ao and f (Ui) Uo :*

*The error sets are supposed to contain a particular standard error ele-* ment ? which is to model in nite computation (divergence). We additionally require (d; ?o) 2 f whenever there is a non-terminating (in nite) computation of f starting with d 2 D .

*i*

*4.1 Correct Implementation*

*Let f*

*be source and f target program semantics, and let 2 ( Ds \* Dt )*

*s t i i* *i*

*and*

*2 ( Ds \* Dt ) be data representation relations between source and*

*o o o*

*target input and output data. We require both relations, i and o, and their*

*inverses 1 and 1, to be strongly error strict (in both directions).*

*i o*

*fs*

*s s*

*D*

*D*

*i o*

*i o*

*Dt Dt*

*i o*

*ft*

*Fig. 2. Source and target program semantics fs; ft and data representations i; o*

*De nition 4.1 (Correct implementation or re nement) ft is said to be* a correct implementation or re nement of fs relative i and o and associated error sets i

*(1) ( i ; ft ) (d) ( fs ; o) (d) [ At*

*o*

*holds for all d 2 Ds with fs (d) Ds [ As (that is with fs (d) \ U s*

*= ;).*

*i o o o*

*For any target program computation, the outcome d00 in Dt is either an* acceptable error output in At , or there exists a source program computation that either ends in a (regular) d0 corresponding to d00 or that ends with an unacceptable (chaotic) error output in U s. That is to say: If ft is a correct

*o*

*o*

*o*

*implementation of fs, then ft either returns a correct result, or an acceptable* error, or, if fs can compute an unacceptable error, ft may (chaotically) return any result.

*If ft correctly implements fs relative i and o, we will write i ; ft w fs ; o* or even shorter just ft w fs (with a boldface w). We indicate this diagram commutativity by the diagram in Figure [3](#_bookmark6) and we can easily prove that we can

*fs*

*s*

*s*

*fs 2 Sems : i o*

*D*

*D*

*i o*

*f 2 Sem :*

*Dt Dt*

*t t i o*

*ft*

*Fig. 3. Commutative diagram expressing correct implementation*

*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]](#_bookmark28) for details).

*It is an important observation, that we can exactly characterize correct* implementation by preservation of relative correctness [[37],](#_bookmark48) which generalizes Floyd's and Hoare's notions of partial respectively total program correctness. Let f 2 ( D \* D ) be a program semantics and let P D and Q D be predicates, i.e., subsets of regular data. f is called (relatively) correct with respect to (pre- and post-conditions) P and Q, h P i f h Q i for short, i

*i o i o*

*(2) f (P ) Q [ Ao*

*Whenever f is started with an input for which the pre-condition P holds,*

*then it either terminates with an output which ful lls the post-condition Q* or else ends with an acceptable error outcome, like for instance a state re- presenting a machine resource violation. The following theorem says that ft correctly implements fs if and only if relative correctness of fs implies relative correctness of ft for all pre- and post-conditions P and Q. A proof can be found in [[17],](#_bookmark28) where we also discuss some variations of the notions de ned here.

*Theorem 4.2 (Preservation of relative correctness) ft correctly imple-*

*ments fs ( i; ft w fs; o ) if and only if for all P Ds and Q Ds we*

*i*

*o*

*have*

*h P i fs h Q i =) h i(P ) i ft h o(Q) i :*

*Relative program correctness generalizes the classical notions of partial or* total program correctness. In order to see this, let f 2 ( Di \* Do ) be an original unextended program semantics and let P Di and Q Do be pre- and post-conditions. f is called partially correct w.r.t. P and Q ( fP g f fQg for short), if f (P ) Q, i.e., if the following holds for f : Whenever the

*precondition P holds for an input d 2 Di and if f (d) Do is de ned, then* any d0 2 f (d) ful lls 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 de ned on any d 2 P , i.e., if f is partially correct w.r.t. P and Q and the domain domf

*of f comprises P (that is if additionally domf P holds).*

*Let us choose particular error sets A = Ai = Ao =def fag and U = Ui =* Uo =def fug with acceptable error element a and unacceptable error element

*u for both domains Di and Do . Hence, = i = o =def A [ U . Note that*

*? 2 fa; ug. We may now consider ? acceptable (a = ? 6= u) or unacceptable*

*(u = ? 6= a). In these special cases, relative correctness of the extended f ext* is equivalent to partial correctness of f

*h P i f ext h Q i () fP g f fQg*

*if ? = a is considered acceptable, and it is equivalent to total correctness, if*

*? = u is considered unacceptable:*

*h P i f ext h Q i () [ P ] f [ Q ] :*

*4.2 Preservation of Partial Correctness*

*Our notion of correct (relative) implementation also generalizes well-known* implementation correctness notions. If we consider ? acceptable in our special situation, then

*(3)*

*ext ; f ext w f ext ; ext*

*() i; ft fs; o*

*i t s o*

*exactly expresses preservation of partial program correctness, and if we con-* sider ? unacceptable, on the other hand, then preservation of relative cor- rectness is equivalent to preservation of total correctness (see again [[17](#_bookmark28)] for details).

*In particular, if we come back to our earlier remarks, preservation of par-* tial program correctness often suÆces 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](#_bookmark40)], or L- simulation [[5])](#_bookmark16) is a practically motivated adequate special case, and according to [(3)](#_bookmark8) above we can easily express it using unextended (partial) relational semantics as follows:

*De nition 4.3 (L-simulation) Let and m denote source and target pro-*

*i*

*o*

*i*

*o*

*grams with semantics [[ ]]*

*SL*

*2 ( DSL*

*\* DSL ) and [[ m ]]*

*2 ( DTL*

*\* DTL ),*

*respectively. Then we say that the implementation step 7! m preserves* partial correctness (or that m L-simulates ), if and only if

*TL*

*(4) i ; [[ m ]] [[ ]] ; o :*

*TL SL*

*That is to say: If the target program m is de ned on the representation*

*of a regular source program input, then the source program can return a*

*corresponding result as well. If and hence m are deterministic, this means* that will return the corresponding result.

*[[ ]] SL*

*2 SemSL :* *i*

*D*

*s*

*i*

*D*

*t*

*[[ ]] SL*

*s*

*D*

*o*

 *o*

*[[ m ]] TL*



*2 SemTL :* *i*

*t o*

*[[ m ]] TL*

*D*

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

*From Theorem* [*4.2*](#_bookmark7) *it is clear that preservation of partial correctness is* equivalent to L-simulation as of the previous de nition:

*Corollary 4.4 (Preservation of partial correctness) An implementation* step 7! m preserves partial correctness if and only if m L-simulates , i.e.,

*8 P; Q : ( f P g f Q g =) f i(P ) g m f o(Q) g ) () i ; [[ m ]] [[ ]] ; o*

*TL SL*

*Preservation of relative correctness gives us the necessary means to de-*

*ne 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 ful ll additional optimization (total correctness) conditions (do not cause particular runtime errors) for the target program to run trustworthily [[17].](#_bookmark28)

*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](#_bookmark35)].

*In* [*[9,10*](#_bookmark21)*] 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](#_bookmark20)] we de ne 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*](#_bookmark20)*] has been on that kind of (machine) program execution. We actually proved* formally and rigorously that source level veri cation is not suÆcient to nally 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](#_bookmark21)].

*The source language SL is a subset of ACL2 Lisp similar to the language* L3 of rst order mutually recursive functions as of [[26],](#_bookmark37) but with s-expressions s 2 SExpr as (dynamic) data. A program is a list defs of function de nitions, followed by a list vars of input variables and a main program expression.

*The (operational) semantics [[ ]] 2 ( SExpr ! SExpr ) is de ned by an*

*SL*

*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](#_bookmark21)] for more details).

*The target language TL is the code of an abstract stack machine. Its* con guration consists of a code part and a separate state stack, which is a data stack containing Lisp s-expressions. The machine has six machine in- structions 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 [[ prog ]]*

*TL*

*2 ( SExpr*

*! SExpr*

*) of machine programs is a stack-*

*transformation de ned by stepwise executing machine instructions. It is de-*

*ned 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 ter- mination 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.

*(s1;::: ; sn)* *2*

*SExpr*

*(evaluate n)*

*SExpr 3 s*

 *t os t os*

*(sn ;::: ; s1;:: :)* *2*

*Sexpr*

*(execute m n)*

*Sexpr*

*3 (s;:: :)*

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

*We prove that compile-program preserves partial correctness (Figure* [*5).*](#_bookmark10) 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](#_bookmark21)]:

*Theorem 5.1 (Compiler correctness formally in ACL2) (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)))))*

*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 is to prove correctness of expression compi-* lation. The correctness theorem for programs above is, after some technical lemmas, a simple consequence of a similar theorem for correct expression com- pilation, applied to the main expression of the program. For correct expression compilation we prove two theorems simultaneously by induction. The rst is the correctness theorem for forms (or expressions), and the second is the theo- rem 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 ma- chine, executed on a compiled expression (list), is de ned 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 de ned 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 have to prove a lot of lemmas. Most of them are technical, but some* re ect crucial insights necessary to nd the proof. So, for instance, it is inte-

*resting, that we actually have to prove the de nedness of the source code se-* mantics and the correctness of the result of machine execution simultaneously as well. The reason is the conditional. The de nedness properties of the con- ditional inductively depend on the value of the condition. The conditional needs not be strict in both alternatives, and will not be for instance in re- cursive de nitions. The conditional has actually been the challenging case to

*nd the proof, not the function application as the reader might have expected.*

*Function application is just captured by computational induction, whereas the* conditional crucially in uences the proof structure in the large. But we do not want to go into further detail. We refer to [[9,10]](#_bookmark21) 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 veri cation of realistic compilers for practical source programming languages and real target processors [[17].](#_bookmark28) 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]](#_bookmark21) only presents an exercise. But the proof is an interesting exercise, and it contributes to realistic compiler veri cation in two important respects:

*It can be accomplished by the mechanical veri cation of subsequent com-* pilation phases, so for instance (b) to a stack machine with a linear heap store (data re nement), (c) to linear assembly code, and (d) to real binary machine code of a concrete processor [[15](#_bookmark26)].

*In the context of the Veri x project on correct compilers* [*[12,15*](#_bookmark26)[*,18],*](#_bookmark29) *the* proof has been generalized to a mechanical PVS [[34](#_bookmark45)] proof of preservation of partial correctness for a larger imperative source language [[6].](#_bookmark17)

*The correctness of step (b) has already been proved manually, and the me-* chanical proof is nearly completed [[11].](#_bookmark22) And step (c) and (d) are implemented [[15](#_bookmark26)[,16,14]](#_bookmark25) 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 e ort to provide an initial fully veri ed compiler executable that preserves partial source program correctness [[15](#_bookmark26)].

*In the PVS proof we use structural operational semantics and inductive re-* lations 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 induc- tion. In our opinion, this is a good news with respect to proof engineering.

*In both respects we bene t 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 rst 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 Veri x* 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 veri cation can succeed, although it is often considered hard or even* impossible |in particular for large and complex systems. Mechanical proof support can help nding proofs (and errors). It may help to redo (or reuse) proofs after slight modi cations. 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 checka-* bility and to intuitive modeling: First of all, in order to convince our customers it is not only necessary that a mechanical proof nally succeeds. Once it does, the prover should also contribute to a consistent and complete proof documen- tation. 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 suÆcient that a program nally 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 veri cation deals with* programming language semantics, and very often programs mean partial func- tions. In ACL2 we modeled them by error strict total functions, which is kind of tedious and always calls for an additional justi cation. Similarly in PVS we use inductive relations, which are also a bit tedious to handle. In both cases we need informal meta-arguments to justify the formalization. Although we know about the problems, we sometimes wish to have a logic of partial functi- ons, and a supporting theorem prover which is as usable and elaborate as the provers we used so far.

*Acknowledgments*

*We would like to thank our colleagues in the Veri x project, in particu-* lar Axel Dold, Thilo Gaul, Gerhard Goos, Andreas Heberle, Friedrich von Henke, Ulrich Ho mann, Hans Langmaack, Vincent Vialard, Wolf Zimmer- mann. Special thanks to Hans Langmaack, and to Markus Muller-Olm and Andreas Wolf for their contributions to the notion of relative program cor- rectness and its preservation. We thank Roy Bartsch for his work and Uwe

*Haferstroh, Sven Nordho and DTK for supporting the work on the Relais* Master case study.

# *References*

*[1] Relais Master Documentation and Manuals. DTK - Gesellschaft fur technische* *Kommunikation MBH, Palmaille 82, Hamburg, Germany, 1996.*

*[2] R. Bartsch and W. Goerigk. Mechanical a-posteriori Veri cation of Results: A Case Study for a Safety Critical AI System (Abstract). In Lina Khatib, editor, Proceedings AAAI Spring Symposium on Model Based Validation of Intelligence* *MBVI'2001, Stanford, CA, U.S.A., March 2001.*

*[3] Roy Bartsch. Mechanisch veri zierte Programmprufung fur die Korrektheit von Prufplanen in der Bahntechnik. Master's thesis, Institut fur 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 Cryptography,* *1989.*

*[5] W.-P. de Roever and K. Engelhardt. Data Re nement: 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.*

*[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 veri ed implementation of an applicative language with* *dynamic storage allocation. PhD thesis, University of Texas at Austin, 1992.*

*[9] W. Goerigk. Compiler Veri cation 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-Universitat zu Kiel, May 2000. To be published.*

*[12] W. Goerigk, A. Dold, Th. Gaul, G. Goos, A. Heberle, F.-W. von Henke,*

*U. Ho mann, H. Langmaack, H. Pfeifer, H. Ruess, and W. Zimmermann. Compiler Correctness and Implementation Veri cation: The Veri x 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, Link ping, Sweden, 1996.*

*[13] W. Goerigk, Th. Gaul, and W. Zimmermann. Correct Programs without Proof? On Checker-Based Program Veri cation. In R. Berghammer and Y. Lakhnech, editors, Proceedings ATOOLS'98 Workshop on \Tool Support for System Speci cation, Development, and Veri cation", Advances in Computing* *Science, Malente, 1998. Springer Verlag.*

*[14] W. Goerigk and U. Ho mann. Compiling ComLisp to Executable Machine Code: Compiler Construction. Technical Report Nr. 9812, Institut fur* *Informatik, University of Kiel, October 1998.*

*[15] W. Goerigk and U. Ho mann. 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. Ho mann. The Compiling Speci cation from ComLisp to Executable Machine Code. Technical Report Nr. 9713, Institut fur Informatik,* *University of 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 fur Informatik, University of Kiel, 2001.*

*[18] G. Goos and W. Zimmermann. Veri cation of Compilers. In E.-R. Olderog and*

*B. Ste en, 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. Bloom eld, 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. Speci cation and veri cation 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., Austin, Texas, August 1994.*

*[25] H. Lange, R. Moller, and B. Neumann. Avoiding Combinatorial Explosion in Automatic Test Generation: Reasoning about Measurements is the Key. In Proceedings of KI'96 Conference on Arti cial Intelligence, Dresden,* *1996.*

*Springer Verlag.*

*[26] Jacques Loeckx and Kurt Sieber. The Foundations of Program Veri cation* *(Second edition). John Wiley and Sons, New York, N.Y., 1987.*

*[27] J S. Moore. Piton: A veri ed assembly level language. Technical Report 22,* *Comp. Logic Inc, Austin, Texas, 1988.*

*[28] J S. Moore. Piton: A Mechanically Veri ed Assembly-Level Language. Kluwer* *Academic Press, Dordrecht, The Netherlands, 1996.*

*[29] M. Muller-Olm. Three Views on Preservation of Partial Correctness. Technical* *Report Veri x/CAU/5.1, University of Kiel, October 1996.*

*[30] M. Muller-Olm. Modular Compiler Veri cation: A Re nement-Algebraic Approach Advocating Stepwise Abstraction, volume 1283 of Lecture Notes in* *Computer Science. Springer-Verlag, Berlin, Heidelberg, New York, 1997.*

*[31] M. Muller-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 Veri cation System. In Deepak Kapur, editor, Proceedings 11th International Conference on Automated Deduction CADE, volume 607 of Lecture Notes in Arti cial* *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 Veri cation", Trento, Italy, 1999.*

*[36] W. Polak. Compiler speci cation and veri cation. In J. Hartmanis and*

*G. Goos, editors, 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 Veri cation. PhD thesis, Technische Fakultat der Christian-Albrechts-Universitat, Kiel, 2001.*