# Verification of Embedded Software: Problems and Perspectives ⋆

Patrick Cousot[1] and Radhia Cousot[2]

[1] cole normale sup rieure
D partement d'informatique
45 rue d'Ulm
75230 Paris cedex 05, France
Patrick.Cousot@ens.fr
http://www.di.ens.fr/~cousot/

[2] Laboratoire d'informatique
CNRS & cole polytechnique
91128 Palaiseau cedex, France
rcousot@lix.polytechnique.fr
http://lix.polytechnique.fr/~rcousot/

**Abstract.** Computer aided formal methods have been very successful for the verification or at least enhanced debugging of hardware. The cost of correction of a hardware bug is huge enough to justify high investments in alternatives to testing such as correctness verification. This is not the case for software for which bugs are a quite common situation which can be easily handled through online updates. However in the area of embedded software, errors are hardly tolerable. Such embedded software is often safety-critical, so that a software failure might create a safety hazard in the equipment and put human life in danger. Thus embedded software verification is a research area of growing importance. Present day software verification technology can certainly be useful but is yet too limited to cope with the formidable challenge of complete software verification. We highlight some of the problems to be solved and envision possible abstract interpretation based static analysis solutions.

## 1 Introduction

Since the origin of computer science, software in general, whence embedded software in particular, expands continuously to consume available processor cycles and memory. The exponential complexity growth in VLSI with decreasing or constant costs is therefore accompanied, maybe with a delay of few months or years, by a corresponding proportional growth in software. So an operating system running a large number of applications which crashes on present day computers every 24 hours will crash every 30 minutes within a decade, probably

⋆ This work was supported in part by the RTD project IST-1999-20527 Daedalus of the european IST FP5 programme.

because of software and not hardware faults. If the present software bug rate is preserved or slightly decreased only, but the size of software is multiplied by a factor of ten, then the computer system might even be expected to crash every three minutes. Embedded software is presently simpler than operating systems but complexity is also growing rapidly in this area. Similar failure rates leads to software crashes every few hours, which is hardly acceptable for safety critical systems, even fault tolerant ones [2]. It follows that verification techniques whether formal or informal must scale up in similar proportions, indeed at a much higher rate since the software verification cost is well-known not to be linear in the software size. We highlight some of the problems to be solved and envision possible abstract interpretation based static analysis solutions.

## 2 Formal methods

Formal methods such as theorem proving based deductive methods [133, 115], model checking [32] and program static analysis by abstract interpretation [43] have all had success stories.

The embedded software for the driverless Meteor line 14 metro in Paris was formally designed with the B-method [4, 62]. The 115 000 lines specification written in B compiles into a 87 000 lines ADA program. The correctness proof, using interactive theorem proving, required to handle manually 27 800 proof obligations. For that purpose, 1400 rules had to be added to the prover and proved correct, 900 of which automatically. Since the metro is running, no error was ever claimed to be found in the embedded software nor in its B specification. Indeed all errors, if any, could only be found at the interfaces, the specification of which might not have been satisfied by the central control software (not developped in B and itself potentially subject to errors). One may wonder why, after such a successful experience, theorem proving based formal methods are not standard for the design of safety critical embedded software. If the circulating figures of 600 person/years are not exagerated this might be because of the human cost of the software development process.

The Ariane 5 flight 501 failure was due to the inertial reference system sending incorrect data following a software exception. This overflow exception was caused by an unprotected data conversion from a too large 64-bit floating point to a 16-bit signed integer value. Not all such conversions were protected because a maximum workload target of 80% had been set for the inertial reference system computer. Ironically, the exception was lifted in a part of the software which serves no purpose after the Ariane 5 launcher lifts off (but was previously required for Ariane 4). An erroneous reasoning based upon physical limitations and large margins of safety lead to the decision to leave variables unprotected. Unfortunately, the overflow was caused by an unexpected high value because the early part of the trajectory of Ariane 5 differs from that of Ariane 4 and results in considerably higher horizontal velocity values. The exception caused the inertial reference system processor to shut down which finally proved fatal [119]. The origin of the error was caught (afterwards) by an abstract interpretation based [46, 47, 50] static analysis of the program [111]. Unfortunately, automatic static analysis relies on approximation, so not all software errors will ever be caught statically in this way. There always exists an approximation to prove a given specification of a given computer system semantics/model but discovering this abstraction is logically equivalent to a formal correctness proof [44]. So one either has to manually design the abstraction (often in the hidden form of a model) or to consider general-purpose reusable abstractions which will always be too abstract to proof some peculiar functional specification.

The most industrialized of the formal methods is certainly model checking [29, 129]. After the famous FDIV design fault in the Pentium processor, most hardware design companies now have model checkers [12, 26]. Present-day hardware model checkers can verify circuit designs of a few hundreds of registers (with abstraction of their surrounding environment). Model checking proceeds by exhaustive enumeration of the state space and is therefore subject to state space explosion: although the checking algorithm may be linear in the size of the specification formula and that of the state space, the state space size often grows exponentially with the size of the description of the model (usually given in the form of a program in some computer language). Despite various symbolic representation techniques using BDDs [24] and their numerous variants, symmetry reduction [30], modular decomposition [107], breadth-first checking with the SAT procedure [16] etc. model checking still has to scale up for hardware, not speaking of software. Difficulties also come out of the temporal logic used for the specification which is often beyond human understanding capabilities [110, 121]. Most of the success of model checking is not so much in the formal verification of refined functional specifications (always subjects to errors in the design of the model and/or specification) but in the finding of bugs not found by other informal methods (such as testing or simulation). Such partial model checking techniques only explore part of the state space (testing or simulation do follow exactly the same principle) thus avoiding the exploration (see e.g. the random pruning of the search space in [96]). Debugging is done at the price of soundness, which is considered abusive by some, practical by others and sometimes is misunderstood.

Despite all these successes, debugging, simulation and run-time tests (using redundant computations to detect faulty numerical computations or to check at run-time that the path traversed is legal) are still the essential computer aided methods in embedded software validation and verification. So the present success of formal methods (mainly in hardware design) is still problematic to scale up for software.

## 3 Challenges in Embedded Software Verification

### 3.1 Software Models

**Programming Language Semantics** Standard models in software are called semantics [3]. They formalize program execution in abstract mathematical terms.

Obviously a programming language semantics can serve as a basis for the analysis and verification of software written in this language. In practice, there are nevertheless many difficulties. Even if (informal) standards do exist (see e.g. ANSI C [102]), most compilers do not strictly implement these specifications. Moreover the standards are continuously revised [101, 103, 104, 102]. An example of change in [103] is "An array subscript is out of range, even if an object is apparently accessible with the given subscript (as in the lvalue expression `a[1][7]` given the declaration `int a[4][5]`) (6.3.6)." Obviously the (probably erroneous) behavior of programs may be completely modified by such an update of their semantics! Programming environments also include many large libraries which semantics is often only very informally specified. Consequently the semantics of a programming language is often that specified by a compiler on a given machine for specific libraries, which is hardly understandable. In the best case, the consequence of this situation is that program verification tools try to conform to standards and therefore do not fully conform to practice. Nevertheless formal tools based upon programming (or specification) language semantics (or an abstract interpretation of this semantics) have the obvious advantage of providing automatically a model of the program to be verified.

**Problem Driven Abstractions for Model Checking** In model-checking, the model is assumed to be given and the verification is relative to that model [34]. The model should preserve only selected characteristics of a real-world artifact, while suppressing others so as to abstract away from the too complex real-world system or program. This abstraction is done informally (or uses abstract interpretation of an already existing more refined model). The requirement to design a model to enable program verification leads to three different descriptions of the real-world system or program: – 1 – in a programming language for the implementation; – 2 – in a verification language for the model and – 3 – in a logic language for the specification of the properties of the model which have to be checked [94]. So the specification is valid for the implementation only if the model is faithful, which is seldom checked (but could be using abstract interpretation to prove the model to be an abstraction of the implementation semantics). Abstraction is sometimes considered in model checking [35, 33, 36], but this is often between a concrete model and a more abstract model thus requiring at least a fourth level in the abstract description of the implementation. Often the concrete model is already assumed to be finite (although too large to be automatically checked) so that the abstraction and concretization functions are now computable. In this context refinement is computable [31], which is not the case in general for the semantics of usual programming languages [80, 81].

**Abstraction** More generally the concrete model is not finite, at least if the most concrete model is considered to be the semantics of the implementation and this implementation is described by programs written in a realistic programming language. The question is then whether the abstract model should be finite or infinite. For the verification of a given infinite concrete model, a finite model will always be adequate [44]. This leads to the idea of automatizing the design of the abstract model from the concrete one, using deductive methods to prove its soundness (e.g. [137, 14, 39, 60, 84, 128, 37, 133, 147]) The difficulty is then that the discovering of the abstraction is logically equivalent to the discovery of an inductive argument (e.g. an invariant) and that the proof that the abstraction is sound is logically equivalent to an inductive proof (e.g. through invariance verification conditions) [44]. Otherwise stated the correctness of the concrete model can always be established by checking a finite abstract model, but the discovery and proof of soundness of the required abstraction is logically equivalent to a direct correctness proof of this concrete model [44]. It is hoped that this will globally simplify the proof (because abstractions like partitioning will decompose the global proof into many local ones [41]). Unfortunately, the soundness proof of the global/local abstractions (which is undecidable) is much more difficult than checking the abstract model (which is finite). The whole difficulty is now in the choice (and soundness proof, if any) of the abstraction so that the benefit is not always clear. Moreover, the whole abstraction/checking process has to be redone after each modification of the program. This is certainly a difficulty for embedded software which often evolves slowly over a long period of times (sometimes up to 20 years). It is therefore necessary to anticipate how the model will be maintained and modified along with the program.

**Standard Abstractions for Program Analysis** For model checkers, the initial abstraction out of the embedded software is provided in the form of an often finite model for a given program. In static program analysis, the model of the program to be verified and its abstraction are provided by the analyzer and proved correct for a given programming language. So the user does not have to extract a verification model from his program but only to choose among predefined abstractions. Since analyzers must work for infinitely many programs, it is shown in [54] that no finite abstraction will be as powerful as infinite abstract domains with widening/narrowing [46, 47] for Turing equivalent programming languages. A broader class of general-purpose abstractions, implemented in the form of libraries, is needed. The elimination of false alarms through the automatic choice of the appropriate abstract domain is still opened.

**Widening/Narrowing and Their Duals** Infinite abstract domains not satisfying chain conditions do require the use of widening/narrowing techniques [46, 47] in order to accelerate the convergence of fixpoint computations into approximations from above or to choose among alternatives in absence of a best approximation. Dual notions do exist for approximations from below [40]. Widening/narrowing techniques are also used in model checking although the link with these well-known techniques is not always recognized (e.g. compare the widening for BDDs of [114] to that of [124, 123]).

The widening/narrowing technique is a dynamic approximation technique (during fixpoint computation) whereas abstraction is a static one (before fixpoint computation, at the time the model/abstract semantics is designed) [47, 20]. All abstractions can be expressed as widenings [52] so abstraction is required only to ensure the existence of an efficient computer-representation of the properties in static analysis and of an initial model in model checking. Otherwise, one can always represent properties as terms although this is not quite adequate in practice since powerful widenings are based on the semantics and the geometry of the fixpoint computation. Widenings based on thresholds (for example the widening to a finite domain [90] in static analysis or the limitation of reachability at a certain depth in model-checking [16 , 1]) are equivalent to static abstraction so are not very expressive [52]. Dynamic widenings could be better exploited in model checking to cope with the state space explosion problem, the same model being explored several times at different levels of abstractions determined dynamically by widenings.

### 3.2 Specifications

The specification language in model checking is typically a temporal logic [29] or a fixpoint calculus [108]. In program analysis, the specification is either provided automatically (e.g. a standard example is absence of run-time errors [50 , 41]) or provided by the user for abstract testing [19 , 41 , 57 , 76]. In both cases, the forward/backward and least/greatest fixpoints based static analysis/checking methods are not so different. Since the design of a model for a program is an abstraction in the sense of abstract interpretation, we can establish the following comparison:

| | | Specification | |
|---|---|---|---|
| | | Program-dependent | Language dependent |
| Abstraction | Program-dependent | Model checking | — |
| | Language dependent | Abstract testing | Static Analysis |

Obviously, one can also think of *Static Checking* where a program-dependent model is designed to check for language dependent properties for which standard abstractions may be a problem (such as threads must eventually enter/exit critical sections, the condition in monitors will eventually be verified for condition variables, etc.).

### 3.3 Control Structures

The flat modelling of control structures by transition systems initially considered in program analysis [50] and model-checking [29 , 129] is valid for some programming languages (like Prolog [53]) but this remains an exception (e.g. for functional languages [56 , 25]). In this context the finiteness hypothesis on data structures is not enough to ensure the finiteness of the program semantics. An example is the restriction of program variables to booleans in which case it is possible to simulate a Turing machine in Pascal [42] but not in C thus enabling finite model checking [11, 67, 95]. Control analysis may also require a precise data flow analysis e.g. to trace pointers to functions or handlers (see Sec. 3.5).

Even with simple control structures, control abstractions (which consist in isolating a control-flow skeleton which is void of any knowledge about data [96]), in particular of the veracity of tests and of run-time errors, is very rough and usable only for safety properties. An example of erroneous reasoning based on this abstraction is live variable in dataflow analysis [138, 58] which is a liveness property for the control-flow model but not for the original program so that the analysis determines potentially live variables only (whence dead variables for sure). So deductive methods or model checking methods for proving liveness of the model while ignoring the program control flow (even partially e.g. by ignoring a single test) perform abstractions from above which are valid for safety but not liveness properties. For such upper abstraction models, most of the power of temporal logics over traditional program analysis methods is simply ruled out. Obviously, the dual notion of abstraction from below can also be used [36] (or both can be mixed [98]) but such lower approximation models are hardly usable to prove more than one property at a time so that different models are needed for proving different liveness properties of the same given program (alternative approaches are discussed in Sec. 3.8).

Although this might be still unfrequent, embedded software will certainly evolve towards multithreaded programming which requires both a high level of expertise together with precise analysis tools to cope with the usual accompanying control flow problems such as untrapped exceptions, race conditions, deadlocks, priority inversion, nonreentrant software, etc.

### 3.4 Numerical Properties

**Integer Properties** The first abstractions into non-relational infinite domains [46, 47] where designed to handle properties of integers. Non-relational numerical abstraction were rapidly followed by relational ones [59]. Such relational domains do scale up for static analysis provided the number of values which can be related is limited either statically at abstraction time [125] or dynamically using widenings [55].

Relational numerical abstract domains with widening have been extensively used in model checking of infinite state spaces to handle safety properties using exactly classical static analysis techniques [86, 63, 85, 88, 89, 100, 109].

For liveness properties, the techniques used in static analysis (inference of variant functions) and in model checking of finite systems (fixpoint approximation from below) are quite different (see Sec. 3.8 below). This is because in the context of infinite state spaces the only dual widenings which are known are based upon variant functions or on the finite prefix/suffix/intermediate exploration of a finite subset of the execution traces (which is nothing but debugging). More work is needing on that subject to cope with liveness properties of embedded software involving integer computations.

**Floating Point Properties** Most present embedded software now involve floating point computations (e.g. to control a trajectory) which used to be performed with fixed precision. A consequence is the uncontrolled loss of precision of the floating-point operations. Transcendental numbers (like $\pi$ and $e$) cannot be represented exactly in a computer, since machines only use finite implementations of numbers (floating-point numbers instead of mathematical real numbers); they are truncated to a given number of decimals. Moreover the usual algebraic laws (associativity for instance) are no longer true when manipulating floating-point numbers. This leads to bugs such as run-time errors (here for instance, uncaught numerical exceptions), but also more subtle ones about the relevance of the numerical calculations that are made which in some cases can be completely non-significant. Let us just take an example reported in [83] showing the importance of the loss of precision. On the 25th of February 1991, during the Gulf war, a Patriot anti-missile missed a Scud in Dharan which in turn crashed onto an American barracks, killing 28 soldiers. The official enquiry report (GAO/IMTEC-92-26) attributed this to a fairly simple "numerical bug". An internal clock that delivers a tick every tenth of a second controlled the missile. Internal time was converted in seconds by multiplying the number of ticks by $\frac{1}{10}$ in a 24 bits register. But $\frac{1}{10} = 0.0001100110011001100110011 00\cdots$ in binary format, i.e. is not represented in an exact manner in memory. This produced a truncating error of about 0.000000095 (decimal), which made the internal computed time drift with respect to ground systems. The battery was in operation for about 100 hours which made the drift of about 0.34 seconds. A Scud flies at about 1676m/s, so the clock error corresponded to a localization error of about 500 meters. The proximity sensors supposed to trigger the explosion of the anti-missile could not find the Scud and therefore the Scud fell and hit the ground, exploding onto the barracks.

Sophisticated semantics and history based abstractions are needed to statically analyze the origin (not only the consequences) of this loss of precision in numerical programs [83]. Note that this is completely different from the boolean verification of circuits in floating-point unit by model checking [28].

### 3.5 Data Structures

In model checking program data structures are most often simply ignored. However the analysis of message-passing transition systems (e.g. for communication protocols) must take message-passing queues and operations into account (often not their content), see e.g. [18]. The abstraction process, which is an abstract interpretation, often remains quite informal or on purely syntactic bases [96], which is not adequate for liveness properties (as noted in previous Sec. 3.3).

Embedded software is often written in C or ADA and uses data structures which cannot be completely ignored when verifying their correctness. An example is the encoding of control into booleans (e.g. when compiling synchronous programs to C) or enumerated types. Type casts may also have to be taken into account. A more complex example is the use of pointers, in the simplest case to pass parameters to procedures (e.g. pointers to buffers, queues, etc.) which may yield to aliases. Any analysis or correctness proof not taking aliases into account would be incorrect. Such pointer alias analysis attempts to determine when two pointer expressions refer to the same storage location and is useful to detect potential side-effects through assignment and parameter passing [48, 49 , 66 , 79 , 92, 93 , 97 , 99 , 112 , 117, 118 , 130 , 132 , 145] (see an overview in [134]). Such memory allocated data structures are used to memorize information which must be traced in some way or another in the correctness analysis or proof. It is then necessary to study the shape of the data structures and the absence of errors in their manipulation (see e.g. [65 , 135, 136]). A classical example of error is buffer overflow (which has been often used by attackers of operating systems). Using precise domains, it is possible to check the absence of overflows with a very low rate of false alarms [64]. Standard abstractions for data structures remain to be developped, e.g. for standard libraries.

### 3.6 Modularity

Modularity has been studied both in model checking and static analysis. Whereas the modules are often designed manually in model checking [107 , 87 , 142], they often follow the modular structure of the software in program static analysis. Four basic methods for compositional separate modular static analysis of programs by abstract interpretation are known [45]: – 1 – Simplification-based separate analysis (where the equations/constraints to be solved for a module are simply simplified while the fixpoint computation is delayed until the context of use of the module is known, see e.g. [77]); – 2 – Worst-case separate analysis (which consists in considering that absolutely no information is known on the interfaces of the module as in the escape analysis of *higher-order* functions by [17] orin the detection of all potential interactions between the agents of a part of a mobile system interacting with an unknown context [74]) ; – 3 – Separate analysis with (user-provided) interfaces (where the properties of the external objects referenced in the program part are defined by the user so that the analysis of the module can rely on that information while any use of the module must guarantee its veracity); and – 4 – Symbolic relational separate analysis (where the analysis of the module relates symbolically the local information within the module to named external objects through a relational domain as in the pointer analysis of [49, Sec. 4.2.2]). There is also a fifth category which is essentially obtained by iterative composition of the above separate local analyses and global analysis methods [45]. For example, very large software based on a library of elementary functions could be analyzed efficiently by a very precise separate (thus possibly parallel) analysis of the basic functions later reused, maybe at a lower degree of precision, for the whole program analysis [45].

### 3.7 Timing

Embedded software (in particular when design according to the model of synchronous languages such as Lustre [27] or Signal [15]), must be shown to satisfy timing constraints (typically execution of all simultaneous "instantaneous" actions must take less than a given upper bound, typically of few milliseconds). Modelling such timing constraints is difficult if not impossible when bounds are tight so that characteristics of modern computers such as pipelines and cache hierarchies must be taken into account. These are numerous extensions of model-checking to handle time (such as timed automata e.g. [7, 141, 6, 21]) but it would be very difficult to manually design appropriate models at the required fine grain level. Indeed in program analysis, the timing semantics can hardly be designed at the programming language source level (for which automatic concrete complexity analysis is certainly useful [75] but insufficient since constants factors do matter [13]!). In practice it is indispensable to consider the program semantics at the assembler level, that is for a given compiler and for a given processor with some hypotheses on the frequency of physical interrupts [144, 5, 69, 70, 68, 143, 139]. The model being automatically generated for the program, one can be confident in its correctness which is established at the assembler language level using a timed model of the considered processor (which is a difficult task). To handle loops [122], one must have an upper-bound on the number of iterations, i.e. handle termination.

### 3.8 Termination and Unbounded Liveness Properties

Although embedded software must usually be proved not terminate except maybe through an operator imperative interaction (which is an easily checked property through reachability), parts of the software (such as elementary loops in basic functions) must be proved to effectively terminate. This is liveness proof which is often much more difficult than safety proofs.

This is not so much the case for finite models, even with fairness hypotheses [34], since in that case the model itself is a safety property (since no loop can go through infinitely many different states) so that any liveness property of the model can be proved by proving a stronger safety property of the model.

However infinite models (e.g. traces generated by an infinite transition system) are usually not safety properties so that proofs much resort to *variant functions* [78, 120] which are much harder to discover than invariants (since they are abstractions of traces not of sets of states [51]). The models considered in model-checking (such as timed automata [7, 141, 6, 21]) are often too restricted to serve as a basis for a general approach.

The results obtained in program analysis, in particular in the context of partial evaluation [116, 9] or for the termination of imperative [105, 113, 106] or functional [127, 23, 22, 82] or logic/constraint [140, 10, 38, 61, 126, 146] languages seems promising. However fairness and schedulers still have to be considered e.g. for infinite state distributed programs in a local network.

### 3.9 Distribution and Mobility

The evolution of critical real-time embedded software for avionics, communication, defense, automotive, utilities, space or medical industry is from centralized control to distributed control on a (e.g. Ethernet-based) local area network (LAN). For example, modern automotive, aeronautic and train transportation computer systems certainly contain or will soon contain several dozen of computers communicating on a LAN. This radically changes the programming models which are presently used, in particular from shared-memory to message-based systems. In this context one must reason on sets of traces (such as UML sequence diagram [91]) and not on sequences of sets of states (as is implicit in most temporal specifications [58]) for which present day set-based abstractions may be inadequate [131]. Experience in this analysis of network protocols [87, 8, 33] is certainly useful but regularity will certainly not be the rule.

Embedded software on LANs will certainly be fully integrated within wide-area networks (such as Internet) before the end of the present decade. For example to meet the constraints resulting from continuous air traffic growth, the future air navigation systems will certainly replace the existing air traffic control systems by effective traffic management that relies on flight path negotiations between the ground and the aircraft to reduce pilot and controller workload which implies network communications e.g. through communication satellites. A characteristic of network software is mobility, at least to replace online components by new ones but also as a communication mean, which implies continuous changes in the communication topology. The proof of non-trivial properties of mobile systems of processes definitely involves unbounded recursive structures which are hard to analyze using uniform models where all instances of processes are merged independently of their instances. Relational abstractions with counting are necessary to obtain precise results [73, 72, 74, 71].

### 3.10 User Interfaces

Tools based on formal methods may require a profound understanding of the methods (e.g. all tools are incomplete so that the user will eventually come with questions that the formal tool cannot fully answer in which case the user will want to understand why no definite positive/negative answer is produced, which even for simple formal systems as simple as type systems is sometime very hard). Interfaces of formal software tools with non-specialists of formal methods, in particular to interpret the output in case of uncertainty and to interact with the tool, is also to be considered.

## 4 Conclusion

Formal verification is certainly essential to design safety critical embedded systems. Embedded software validation must evolve from debugging to verification, still a long way to go. The increasing complexity of such software systems evolves with the new capacities of hardware and networking capabilities. This continuously increased complexity makes the verification of embedded software a formidable challenge for the next decade.

The success will certainly depends on which models/semantics and specifications of embedded software are considered:

- Hand-made models are a guarantee of success (since finite models allowing for a formal proof of a given program always exist [44]). However hand-made models are also extremely costly to design (but maybe for relatively small parts of the program) and maintain (in order to follow program evolutions). Methods for proving the soundness of such hand-made models are still to be developped and could strongly rely on abstract interpretation.
- Application specific models constructed by automatic abstraction require a model of reference and a theorem prover based soundness proof. This is logically equivalent to direct formal proofs [44] and so are also likely to be extremely costly (again but for small parts of the program).
- Since abstraction is inevitable and costly, an interesting alternative is to design reusable abstractions for programming languages with respect to predefined specifications (as in static program analysis) or user-defined specifications (as in abstract testing [57]). This is the whole purpose of program semantics abstraction as formalized by abstract interpretation [43] which will certainly be an important formal method leading to the systematic design of useful automatic tools supporting the design of embedded software.

## References

[1] P.A. Abdulla, P. Bjesse, and N. E n. Symbolic reachability analysis based on SAT-solvers. In S. Graf and M.I. Schwartzbach, editors, *Proc. 6th Int. Conf. TACAS '2000*, Berlin, DE, 25 Mar. – 2 Apr. 2000, LNCS 1785, pages 411–425. Springer-Verlag, 2000.

[2] J.A. Abraham. The myth of fault tolerance in complex systems, keynote speech. In *The Pacific Rim International Symposium on Dependable Computing, PRDC'99*, Hong Kong, CN. IEEE Comp. Soc. Press, 16–17 Dec. 1999. http://www.cerc.utexas.edu/~jaa/talks/prdc-1999/.

[3] S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors. *Semantic Modelling*, volume 4 of *Handbook of Logic in Computer Science*. Clarendon Press, 1995.

[4] J.-R. Abrial. *The B-Book*. Cambridge U. Press, 1996.

[5] M. Alt, Ferdinand C., F. Martin, and R. Wilhelm. Cache behavior prediction by abstract interpretation. In R. Cousot and D.A. Schmidt, editors, *Proc. 3rd Int. Symp. SAS '96*, Aachen, DE, 24–26 Sep. 1996, LNCS 1145, pages 52–66. Springer-Verlag, 1996.

[6] R. Alur. Timed automata. In N. Halbwachs and D. Peled, editors, *Proc. 11th Int. Conf. CAV '99*, Trento, IT, LNCS 1633, pages 8–22. Springer-Verlag, 6–10 Jul. 1999.

[7] R. Alur and D.L. Dill. A theory of timed automata. *Theoret. Comput. Sci.*, 126(2):183–235, 1994.

[8] R. Alur and B.-Y. Wang. Verifying network protocol implementations by symbolic refinement checking. In G. Berry, H. Comon, and A. Finkel, editors, *Proc. 30th Int. Conf. CAV '01*, Paris, FR, LNCS 2102, pages 169–181. Springer-Verlag, 16–18 Jul. 2001.

[9] P.H. Andersen and Holst C.K. Termination analysis for offline partial evaluation of a higher order functional language. In R. Cousot and D.A. Schmidt, editors,

*Proc. 3$^{rd}$ Int. Symp. SAS '96*, Aachen, DE, 24–26 Sep. 1996, LNCS 1145, pages 67–82. Springer-Verlag, 1996.

[10] J. Bailey, L. Crnogorac, K. Ramamohanarao, and H. Søndergaard. Abstract interpretation of active rules and its use in termination analysis. In F.N. Afrati and P. Kolaitis, editors, *Proc. 6$^{th}$ Int. Conf. ICDT '97*, Delphi, GR, LNCS 1186, pages 188–202. Springer-Verlag, 8–10 Jan. 1997.

[11] T. Ball and S.K. Rajamani. Bebop: A symbolic model checker for boolean programs. In K. Havelund, J. Penix, and W. Visser, editors, *Proc. 7$^{th}$ SPIN Workshop*, Stanford, CA, LNCS 1885, pages 113–130. Springer-Verlag, Aug. 30 – Sep. 1, 2000.

[12] I. Beer, S. Ben-David, C. Eisner, D. Geist, L. Gluhovsky, T. Heyman, A. Landver, P. Paanah, Y. Rodeh, G. Ronin, and Y. Wolfsthal. RuleBase: Model checking at IBM. In O. Grumberg, editor, *Proc. 9$^{th}$ Int. Conf. CAV '97*, Haifa, IL, LNCS 1254, pages 480–483. Springer-Verlag, 22–25 Jul. 1997.

[13] A.M. Ben-Amram and N.D. Jones. Computational complexity via programming languages: constant factors do matter. *Acta Informat.*, 37(2):83–120, 2000.

[14] S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionally and automatically. In A.J. Hu and M.Y. Vardi, editors, *Proc. 10$^{th}$ Int. Conf. CAV '98*, Vancouver, BC, CA,LNCS 1427, pages 319–331. Springer-Verlag, 28 June – 2 Jul. 1998.

[15] A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Sci. Comput. Programming*, 16(2):103–149, 1991.

[16] A. Biere, A. Cimatti, E.M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proc. 36$^{th}$ Conf. DAC '99*, New Orleans, LA, pages 317–320. ACM Press, 21–25 June 1999.

[17] B. Blanchet. Escape analysis: Correctness proof, implementation and experimental results. In *25$^{th}$ POPL*, pages 25–37, San Diego, CA, 19–21 Jan. 1998. ACM Press.

[18] B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs (extended abstract). In R. Alur and T.A. Henzinger, editors, *Proc. 8$^{th}$ Int. Conf. CAV '96*, New Brunswick, NJ, LNCS 1102, pages 1–12. Springer-Verlag, 31 Jul. –3 Aug. 1996.

[19] F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *Proc. ACM SIGPLAN '93 Conf. PLDI. ACM SIGPLAN Not. 28(6)*, pages 46–55, Albuquerque, NM, 23–25 June 1993. ACM Press.

[20] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In D. Bjørner, M. Broy, and I.V. Pottosin, editors, *Proc. FMPA*, Akademgorodok, Novosibirsk, RU, LNCS 735, pages 128–141. Springer-Verlag, 28 June –2 Jul. 1993.

[21] M. Bozga, O. Maler, A. Pnueli, and S. Yovine. Some progress in the symbolic verification of timed automata. In O. Grumberg, editor, *Proc. 9$^{th}$ Int. Conf. CAV '97*, Haifa, IL, LNCS 1254, pages 179–190. Springer-Verlag, 22–25 Jul. 1997.

[22] J. Brauburger. Automatic termination analysis for partial functions using polynomial orderings. In P. Van Hentenryck, editor, *Proc. 4$^{th}$ Int. Symp. SAS '97*, Paris, FR, 8–10 Sep. 1997, LNCS 1302, pages 330–344. Springer-Verlag, 1997.

[23] J. Brauburger and J. Giesl. Termination analysis for partial functions. In R. Cousot and D.A. Schmidt, editors, *Proc. 3rd Int. Symp. SAS '96*, Aachen, DE, 24–26 Sep. 1996, LNCS 1145, pages 113–127. Springer-Verlag, 1996.

[24] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: $10^{20}$ states and beyond. *Inform. and Comput.*, 98(2):142–170, June 1992.

[25] G.L. Burn, C.L. Hankin, and S. Abramsky. The theory of strictness analysis for higher-order functions. In H. Ganzinger and N.D. Jones, editors, *Programs as Data Objects, Proceedings of a Workshop*, Copenhagen, DK, 17–19 Oct. 1985, LNCS 217, pages 42–62. Springer-Verlag, 1986.

[26] Cadence®. "formalcheck" model checking verification. `http://www.cadence.com/datasheets/formalcheck.html`.

[27] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. Lustre: a declarative language for programming synchronous systems. In *14th POPL*, Munchen, DE, 1987. ACM Press.

[28] Y-A. Chen, E.M. Clarke, P.H. Ho, Y. Hoskote, T. Kam, M. Khaira, J. O'Leary, and X. Zhao. Verification of all circuits in a floating-point unit using word-level model checking. In M.S. Srivas and A.J. Camilleri, editors, *Proc. 1st Int. Conf. on Formal Methods in Computer-Aided Design, FMCAD '96*, number 1166 in LNCS, pages 19–33, Palo Alto, CA, 6–8 Nov. 1996. Springer-Verlag.

[29] E.M. Clarke and E.A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *IBM Workshop on Logics of Programs*, Yorktown Heights, NY, US, LNCS 131. Springer-Verlag, May 1981.

[30] E.M. Clarke, E.A. Emerson, S. Jha, and A.P. Sistla. Symmetry reductions in model checking. In A.J. Hu and M.Y. Vardi, editors, *Proc. 10th Int. Conf. CAV '98*, Vancouver, BC, CA,LNCS 1427, pages 147–158. Springer-Verlag, 28 June – 2 Jul. 1998.

[31] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and . Veith. Counterexample-guided abstraction refinement. In E.A. Emerson and A.P. Sistla, editors, *Proc. TWELFTHInt. Conf. CAV '00*, Chicago, IL, LNCS 1855, pages 154–169. Springer-Verlag, 15–19 Jul. 2000.

[32] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and . Veith. Progress on the state explosion problem in model checking. In R. Wilhelm, editor, *« Informatics — 10 Years Back, 10 Years Ahead »*, volume 2000 of *LNCS*, pages 176–194. Springer-Verlag, 2000.

[33] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *TOPLAS*, 16(5):1512–1542, Sep. 1994.

[34] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking.* MIT Press, 1999.

[35] E.M. Clarke, S. Jha, Y. Lu, and D. Wang. Abstract BDDs: A technique for using abstraction in model checking. In L. Pierre and T. Kropf, editors, *Correct Hardware Design and Verification Methods, Proc. 10th IFIP WG 10.5 Adv. Res. Work. Conf. CHARME '99*, Bad Herrenalp, DE, LNCS 1703, pages 172–186. Springer-Verlag, 27–29 Sep. 1999.

[36] R. Cleaveland, P. Iyer, and D. Yankelevitch. Optimality in abstractions of model checking. In A. Mycroft, editor, *Proc. 2nd Int. Symp. SAS '95*, Glasgow, UK, 25–27 Sep. 1995, LNCS 983, pages 51–63. Springer-Verlag, 1995.

[37] W.R. Cleaveland, editor. *On Proving Safety Properties by Integrating Sta-*

*tic Analysis, Theorem Proving and Abstraction*, Amsterdam, NL, LNCS 1579. Springer-Verlag, 22–28 Mar. 1999.

[38] M. Codish and C. Taboch. A semantic basis for the termination analysis of logic programs goal. *J. Logic Programming*, 41(1):103–123, 1999.

[39] M.A. Col n and T.E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In A.J. Hu and M.Y. Vardi, editors, *Proc. $10^{th}$ Int. Conf. CAV '98*, Vancouver, BC, CA, LNCS 1427, pages 293–304. Springer-Verlag, June /Jul. 1998.

[40] P. Cousot. *M thodes it ratives de construction et d'approximation de points fixes d'op rateurs monotones sur un treillis, analyse s mantique de programmes.* Th se d' tat s sciences math matiques, Universit scientifique et m dicale de Grenoble, Grenoble, FR, 21 Mar. 1978.

[41] P. Cousot. Semantic foundations of program analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, 1981.

[42] P. Cousot. Methods and logics for proving programs. In J. van Leeuwen, editor, *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, chapter 15, pages 843–993. Elsevier, 1990.

[43] P. Cousot. Abstract interpretation based formal methods and future challenges, invited paper. In R. Wilhelm, editor, *« Informatics — 10 Years Back, 10 Years Ahead »*, volume 2000 of *LNCS*, pages 138–156. Springer-Verlag, 2000.

[44] P. Cousot. Partial completeness of abstract fixpoint checking, invited paper. In B.Y. Choueiry and T. Walsh, editors, *Proc. $4^{th}$ Int. Symp. SARA '2000*, Horseshoe Bay, TX, LNAI 1864, pages 1–25. Springer-Verlag, 26–29 Jul. 2000.

[45] P. Cousot. Compositional separate modular static analysis of programs by abstract interpretation. *Proc. SSGRR 2001 – Advances in Infrastructure for Electronic Business, Science, and Education on the Internet*, 6 – 10 Aug. 2001.

[46] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proc. $2^{nd}$ Int. Symp. on Programming*, pages 106–130. Dunod, 1976.

[47] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *$4^{th}$ POPL*, pages 238–252, Los Angeles, CA, 1977. ACM Press.

[48] P. Cousot and R. Cousot. Static determination of dynamic properties of generalized type unions. In *ACM Symposium on Language Design for Reliable Software*, Raleigh, NC, ACM SIGPLAN Not. 12(3):77–94, 1977.

[49] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In E.J. Neuhold, editor, *IFIP Conf. on Formal Description of Programming Concepts, St-Andrews, N.B., CA*, pages 237–277. North-Holland, 1977.

[50] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *$6^{th}$ POPL*, pages 269–282, San Antonio, TX, 1979. ACM Press.

[51] P. Cousot and R. Cousot. ' la Floyd' induction principles for proving inevitability properties of programs. In M. Nivat and J. Reynolds, editors, *Algebraic Methods in Semantics*, chapter 8, pages 277–312. Cambridge U. Press, 1985.

[52] P. Cousot and R. Cousot. Comparison of the Galois connection and widening/narrowing approaches to abstract interpretation. *Actes JTASPEFL '91, Bor-*

*deaux, FR. BIGRE*, 74:107–110, Oct. 1991.

[53] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *J. Logic Programming*, 13(2–3):103–179, 1992. (The editor of J. Logic Programming has mistakenly published the unreadable galley proof. For a correct version of this paper, see `http://www.di.ens.fr/~cousot`.).

[54] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper. In M. Bruynooghe and M. Wirsing, editors, *Proc. $4^{th}$ Int. Symp. PLILP '92*, Leuven, BE, 26–28 Aug. 1992, LNCS 631, pages 269–295. Springer-Verlag, 1992.

[55] P. Cousot and R. Cousot. Galois connection based abstract interpretations for strictness analysis, invited paper. In D. Bjørner, M. Broy, and I.V. Pottosin, editors, *Proc. FMPA*, Akademgorodok, Novosibirsk, RU, LNCS 735, pages 98–127. Springer-Verlag, 28 June – 2 Jul. 1993.

[56] P. Cousot and R. Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages), invited paper. In *Proc. 1994 ICCL*, pages 95–112, Toulouse, FR, 16–19 May 1994. IEEE Comp. Soc. Press.

[57] P. Cousot and R. Cousot. Abstract interpretation based program testing, invited paper. In *Proc. SSGRR 2000 Computer & eBusiness International Conference*, Compact disk paper 248 and electronic proceedings `http://www.ssgrr.it/en/ssgrr2000/proceedings.htm`, L'Aquila, IT, 31 Jul. – 6 Aug. 2000. Scuola Superiore G. Reiss Romoli.

[58] P. Cousot and R. Cousot. Temporal abstract interpretation. In *$27^{th}$ POPL*, pages 12–25, Boston, MA, Jan. 2000. ACM Press.

[59] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *$5^{th}$ POPL*, pages 84–97, Tucson, AZ, 1978. ACM Press.

[60] S. Das, D.L. Dill, and S. Park. Experience with predicate abstraction. In N. Halbwachs and D. Peled, editors, *Proc. $11^{th}$ Int. Conf. CAV '99*, Trento, IT, LNCS 1633, pages 160–171. Springer-Verlag, 6–10 Jul. 1999.

[61] S. Decorte, D. De Schreye, and H. Vandecasteele. Constraint-based termination analysis of logic programs. *TOPLAS*, 21(6):1137–1195, Nov. 1999.

[62] B. Dehbonei and F. Mejia. Formal methods in the railways signalling industry. In M. Naftalin, T. Denvir, and M. Bertran, editors, *Proc. $2^{nd}$ FME '94, Industrial Benefit of Formal Methods*, Barcelona, ES, LNCS 873, pages 26–34. Springer-Verlag, 24–28 Oct. 1994.

[63] D.L. Dill and H. Wong-Toi. Verification of real-time systems by successive over and under approximation. In P. Wolper, editor, *Proc. $7^{th}$ Int. Conf. CAV '95*, Li ge, BE, LNCS 939, pages 409–422. Springer-Verlag, 3–5 Jul. 1995.

[64] N. Dor, M. Rodeh, and M. Sagiv. Cleanness checking of string manipulations in c programs via integer analysis. In P. Cousot, editor, *Proc. $8^{th}$ Int. Symp. SAS '01*, Paris, FR, LNCS 2126, pages 194–212. Springer-Verlag, 16–18 Jul. 2001.

[65] N. Dor, M. Rodeh, and M. Sagiv. Checking cleanness in linked lists. In J. Palsberg, editor, *Proc. $7^{th}$ Int. Symp. SAS '2000*, Santa Barbara, CA, LNCS 1824, pages 115–134. Springer-Verlag, 29 June – 1 Jul. 2000.

[66] M. Emami, R. Ghiya, and L.J. Hendren. Context-sensitive interprocedural

points-to analysis in the presence of function pointers. In *Proc. ACM SIGPLAN '93 Conf. PLDI. ACM SIGPLAN Not. 28(6)*, pages 242–256, Orlando, FL, June 1994. ACM Press.

[67] J. Esparza and S. Schwoon. A BDD-based model checker for recursive programs. In G. Berry, H. Comon, and A. Finkel, editors, *Proc. 30$^{th}$ Int. Conf. CAV '01*, Paris, FR, LNCS 2102, pages 324–336. Springer-Verlag, 16–18 Jul. 2001.

[68] C. Ferdinand, D. K stner, M. Langenbach, F. Martin, M. Schmidt, J. Schneider, H. Theiling, S. Thesing, and R. Wilhelm. Run-time guarantees for real-time systems - the USES approach. In K. Beiersd rfer, G. Engels, and W. Sch fer, editors, *Informatik '99 - Informatik §berwindet Grenzen, 29. Jahrestagung der Gesellschaft f§r Informatik, Paderborn, DE.* Springer-Verlag, Informatik Aktuell, Oct. 1999.

[69] C. Ferdinand, F. Martin, R. Wilhelm, and M. Alt. Cache behavior prediction by abstract interpretation. *Sci. Comput. Programming*, 35(1):163–189, 1999.

[70] C. Ferdinand and R. Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Syst.*, 17(2–3):131–181, 1999.

[71] J. Feret. Occurrence counting analysis for the $\pi$-calculus. In P. Cousot, . Goubault, J. Gunawardena, M. Herlihy, M. Raussen, and V. Sassone, editors, *Preliminary Proc. Workshop GETCO '00*, pages 99–116, State College, 21 Aug. 2000. BRICS Notes Series NS-00-3.

[72] J. Feret. Abstract interpretation-based static analysis of mobile ambients. In P. Cousot, editor, *Proc. 8$^{th}$ Int. Symp. SAS '01*, Paris, FR, LNCS 2126, pages 413–431. Springer-Verlag, 16–18 Jul. 2001.

[73] J. Feret. Occurrence counting analysis for the $\pi$-calculus. *ENTCS*, 39, 2001. `http://www.elsevier.nl/locate/entcs/volume39.html`.

[74] J. Feret. Confidentiality analysis of mobile systems. In J. Palsberg, editor, *Proc. 7$^{th}$ Int. Symp. SAS '2000*, Santa Barbara, CA, LNCS 1824, pages 135–154. Springer-Verlag, 29 June – 1 Jul. 2000.

[75] P. Flajolet, B. Salvy, and P. Zimmermann. Automatic average-case analysis of algorithm. *Theoret. Comput. Sci.*, 79(1):37–109, 1991.

[76] C. Flanagan, M. Flatt S. Krishnamurthi, S. Weirich, and M. Felleisen. Static debugging: Browsing the web of program invariants. In *Proc. ACM SIGPLAN '96 Conf. PLDI. ACM SIGPLAN Not. 31(5)*, pages 23–32, Philadelphia, PA, 21–24, May 1996.

[77] C. Flanagan and J.B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *28$^{th}$ POPL. ACM SIGPLAN Not. 36(3)*, pages 193–205, London, GB, Jan. 2001. ACM Press.

[78] R.W. Floyd. Assigning meaning to programs. In J.T. Schwartz, editor, *Proc. Symposium in Applied Mathematics*, volume 19, pages 19–32. AMS, 1967.

[79] R. Ghiya and L.J. Hendren. Putting pointer analysis to work. In *25$^{th}$ POPL*, pages 121–133, San Diego, CA, 19–21 Jan. 1998. ACM Press.

[80] R. Giacobazzi and E. Quintarelli. Incompleteness, counterexamples and refinements in abstract model-checking. In P. Cousot, editor, *Proc. 8$^{th}$ Int. Symp. SAS '01*, Paris, FR, LNCS 2126, pages 356–373. Springer-Verlag, 16–18 Jul. 2001.

[81] R. Giacobazzi and F. Ranzato. Uniform closures: Order-theoretically reconstructing logic program semantics and abstract domain refinements. *Inform.*

*and Comput.*, 145(2):153–190, 1998.

[82] J. Giesl. Termination analysis for functional programs using term orderings. In A. Mycroft, editor, *Proc. 2nd Int. Symp. SAS '95*, Glasgow, UK, 25–27 Sep. 1995, LNCS 983, pages 154–171. Springer-Verlag, 1995.

[83] . Goubault. Static analyses of the precision of floating-point operations. In P. Cousot, editor, *Proc. 8th Int. Symp. SAS '01*, Paris, FR, LNCS 2126, pages 234–259. Springer-Verlag, 16–18 Jul. 2001.

[84] S. Graf and C. Loiseaux. A tool for symbolic program verification and abstraction. In C. Courcoubetis, editor, *Proc. 5th Int. Conf. CAV '93*, Elounda, GR, LNCS 697, pages 71–84. Springer-Verlag, 28 June –1 Jul. 1993.

[85] N. Halbwachs. Delays analysis in synchronous programs. In C. Courcoubatis, editor, *Proc. 5th Int. Conf. CAV '93*, Elounda, GR, LNCS 697, pages 333–346. Springer-Verlag, 28 June –1 Jul. 1993.

[86] N. Halbwachs. About synchronous programming and abstract interpretation. In B. Le Charlier, editor, *Proc. 1st Int. Symp. SAS '94*, Namur, BE, 20–22 Sep. 1994, LNCS 864, pages 179–192. Springer-Verlag, 1994.

[87] N. Halbwachs, F. Lagnier, and C. Ratel. An experience in proving regular networks of processes by modular model checking. *Acta Informat.*, 29(6/7):523–543, 1992.

[88] N. Halbwachs, J.- . Proy, and P. Raymond. Verification of linear hybrid systems by means of convex approximations. In B. Le Charlier, editor, *Proc. 1st Int. Symp. SAS '94*, Namur, BE, 20–22 Sep. 1994, LNCS 864, pages 223–237. Springer-Verlag, 1994.

[89] N. Halbwachs, Y.E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, Aug. 1997.

[90] C. Hankin and S. Hunt. Approximate fixed points in abstract interpretation. *Sci. Comput. Programming*, 22(3):283–306, 1994. Erratum: Sci. Comput. Programming 23(1): 103 (1994).

[91] Ø. Haugen. From MSC-2000 to UML 2.0 – the future of sequence diagrams. In R. Reed and J. Reed, editors, *Proc. SDL 2001: Meeting UML, 10th Int. SDL Forum*, Copenhagen, DK, 27–29 June 2001, LNCS 2078, pages 38–51. Springer-Verlag, 2001.

[92] M. Hind, M. Burke, P. Carini, and J.-D. Choi. Interprocedural pointer alias analysis. *TOPLAS*, 21(4):848–894, Jul. 1999.

[93] M. Hind and A. Pioli. Assessing the effects of flow-sensitivity on pointer alias analyses. In G. Levi, editor, *Proc. 5th Int. Symp. SAS '98*, Pisa, IT, 14–16 Sep. 1998, LNCS 1503, pages 57–81. Springer-Verlag, 1998.

[94] G.J. Holzmann. From code to models. In *Proc. 2nd Int. Conf. ACSD '01*, Newcastle upon Tyne, GB. IEEEpress, 25–29 June 2001.

[95] G.J. Holzmann. Logic verification of ANSI-C code with SPIN. In K. Havelund, J. Penix, and W. Visser, editors, *Proc. 7th SPIN Workshop*, Stanford, CA, LNCS 1885, pages 131–147. Springer-Verlag, Aug. 30 – Sep. 1, 2000.

[96] G.J. Holzmann and M.H. Smith. Software model checking: Extracting verification models from source code. In *Proc. Formal Methods in Software Engineering and Distributed Systems, PSTV/FORTE99*, Beijng
china, pages 481–497. Kluwer Acad. Pub., Oct. 1999.

[97] S. Horwitz. Precise flow-insensitive may-alias analysis is NP-hard. *TOPLAS*, 19(1):1–6, Jan. 1997.

[98] M. Huth, R. Jagadeesan, and D.A. Schmidt. Modal transition systems: A foundation for three-valued program analysis. In D. Sands, editor, *Proc.* $10^{th}$ *ESOP '01*, LNCS 2028, pages 155–169, Genova, IT, 2–6 Apr. 2001. Springer-Verlag.

[99] S. Jagannathan, P. Thiemann, S. Weeks, and A.K. Wright. Single and loving it: Must-alias analysis for higher-order languages. In $25^{th}$ *POPL*, pages 329–341, San Diego, CA, 19–21 Jan. 1998. ACM Press.

[100] B. Jeannet, N. Halbwachs, and P. Raymond. Dynamic partitioning in analyses of numerical properties. In A. Cortesi and G. Fil , editors, *Proc.* $6^{th}$ *Int. Symp. SAS '99*, Venice, IT, 22–24 Sep. 1999, LNCS 1694, pages 18–38. Springer-Verlag, 1999.

[101] Joint Technical Committee ISO/IEC JTC1, Information Technology. The ISO/IEC 9899:1990 standard for Programming Language C. 1 Dec. 1990.

[102] Joint Technical Committee ISO/IEC JTC1, Information Technology. The ISO/IEC 9899:1999 standard for Programming Language C. 1 Dec. 1999.

[103] Joint Technical Committee ISO/IEC JTC1, Information Technology. The Technical Corrigendum 1 (ISO/IEC 9899 TCOR1) to ISO/IEC 9899:1990 standard for Programming Language C. `http://anubis.dkuug.dk/JTC1/SC22/WG14/www/docs/tc2.htm`, 1995.

[104] Joint Technical Committee ISO/IEC JTC1, Information Technology. The Technical Corrigendum 2 (ISO/IEC 9899 TCOR2) to ISO/IEC 9899:1990 standard for Programming Language C. `http://anubis.dkuug.dk/JTC1/SC22/WG14/www/docs/tc1.htm`, 1996.

[105] N.D. Jones. Program analysis for implicit computational complexity. In 0. Danvy and A. Filinski, editors, *Proc.* $2^{nd}$ *Symp. PADO '2001*, rhus, DK, 21–23 May 2001, LNCS 2053, page 1. Springer-Verlag, 2001.

[106] N.D. Jones. Program termination analysis by size-change graphs (abstract). In R. Gor , A. Leitsch, and T. Nipkow, editors, *Proc.* $1^{st}$ *IJCAR '01*, Siena, IT, LNCS 2083, pages 1–4. Springer-Verlag, 18–23 June 2001.

[107] Y. Kesten and A. Pnueli. Modularization and abstraction: The keys to formal verification. In L. Brim, J. Gruska, and J. Zlatuska, editors, $23^{rd}$ *Int. Symp. MFCS '98*, LNCS 1450, pages 54–71. Springer-Verlag, 1998.

[108] D. Kozen. Results on the propositional $\mu$-calculus. *Theoret. Comput. Sci.*, 27:333–354, 1983.

[109] O. Kupferman and M.Y. Vardi. Model checking of safety properties. In N. Halbwachs and D. Peled, editors, *Proc.* $11^{th}$ *Int. Conf. CAV '99*, Trento, IT, LNCS 1633, pages 172–183. Springer-Verlag, 6–10 Jul. 1999.

[110] O. Kupferman and M.Y. Vardi. Vacuity detection in temporal model checking. In L. Pierre and T. Kropf, editors, *Correct Hardware Design and Verification Methods, Proc.* $10^{th}$ *IFIP WG 10.5 Adv. Res. Work. Conf. CHARME '99*, Bad Herrenalp, DE, LNCS 1703, pages 82–96. Springer-Verlag, 27–29 Sep. 1999.

[111] P. Lacan, J.N. Monfort, L.V.Q. Ribal, A. Deutsch, and G. Gonthier. The software reliability verification process: The ARIANE 5 example. In *Proceedings DASIA 98 – DAta Systems In Aerospace*, Athens, GR. ESA Publications, SP-422, 25–28 May 1998.

[112] W.A. Landi. Undecidability of static analysis. *ACM Lett. Prog. Lang. Syst.*,

1(4):323–337, Dec. 1992.

[113] C.S. Lee, N.D. Jones, and A.M. Ben-Amram. The size-change principle for program termination. In *28th POPL*, pages 81–92, LondonLondon, GB, Jan. 2001. ACM Press.

[114] W. Lee, A. Pardo, J.-Y. Jang, G. Hachtel, and F. Somenzi. Tearing based automatic abstraction for CTL model checking. In *ICCAD 1996*, San Jose, CA, pages 76–81. IEEE Comp. Soc. Press, Nov. 10–14 1996.

[115] K.R.M. Leino. Extended static checking: A ten-year perspective. In R. Wilhelm, editor, *« Informatics — 10 Years Back, 10 Years Ahead »*, volume 2000 of *LNCS*, pages 157–175. Springer-Verlag, 2000.

[116] M. Leuschel. On the power of homeomorphic embedding for online termination. In G. Levi, editor, *Proc. 5th Int. Symp. SAS '98*, Pisa, IT, 14–16 Sep. 1998, LNCS 1503, pages 200–214. Springer-Verlag, 1998.

[117] D. Liang and M.J. Harrold. Efficient points-to analysis for whole-program analysis. In O. Nierstrasz and M. Lemoine, editors, *Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, LNCS 1687*, pages 199–215, Toulouse, FR, Sep. 1999.

[118] D. Liang and M.J. Harrold. Efficient computation of parameterized pointer information for interprocedural analysess. In P. Cousot, editor, *Proc. 8th Int. Symp. SAS '01*, Paris, FR, LNCS 2126, pages 279–298. Springer-Verlag, 16–18 Jul. 2001.

[119] J.L. Lions (Chairman of the Board). Ariane 5 flight 501 failure, report by the inquiry board. `http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html`, see also `http://vlsi.colorado.edu/~abel/pubs/anecdote.html#ariane`.

[120] Z. Manna and A. Pnueli. Axiomatic approach to total correctness. *Acta Informat.*, 3:253–263, 1974.

[121] T. Margaria and W. Yi, editors. *Branching vs. Linear Time: Final Showdown*, Genova, IT, LNCS 2031. Springer-Verlag, 2–6 Apr. 2001.

[122] F. Martin, M. Alt, R. Wilhelm, and C. Ferdinand. Analysis of loops. In K. Koskimies, editor, *Proc. 7th Int. Conf. CC '98*, Lisbon, PT, LNCS 1383, pages 80–94. Springer-Verlag, 28 Mar. – 4 Apr. 1998.

[123] L. Mauborgne. Abstract interpretation using TDGs. In B. Le Charlier, editor, *Proc. 1st Int. Symp. SAS '94*, Namur, BE, 20–22 Sep. 1994, LNCS 864, pages 363–379. Springer-Verlag, 1994.

[124] L. Mauborgne. Abstract interpretation using typed decision graphs. *Sci. Comput. Programming*, 31(1):91–112, May 1998.

[125] A. Min . A new numerical abstract domain based on difference-bound matrices. In 0. Danvy and A. Filinski, editors, *Proc. 2nd Symp. PADO '2001*, rhus, DK, 21–23 May 2001, LNCS 2053, pages 155–172. Springer-Verlag, 2001.

[126] M. M ller, T. Glaß, and K. Stroetmann. Automated modular termination proofs for real Prolog programs. In R. Cousot and D.A. Schmidt, editors, *Proc. 3rd Int. Symp. SAS '96*, Aachen, DE, 24–26 Sep. 1996, LNCS 1145, pages 220–237. Springer-Verlag, 1996.

[127] S.E. Panitz and M. Schmidt-Schauß. TEA: Automatically proving termination of programs in a non-strict higher-order functional language. In P. Van Hentenryck,

editor, *Proc. $4^{th}$ Int. Symp. SAS '97*, Paris, FR, 8–10 Sep. 1997, LNCS 1302, pages 345–360. Springer-Verlag, 1997.

[128] A. Pardo and G.D. Hachtel. Automatic abstraction techniques for propositional $\mu$-calculus model checking. In O. Grumberg, editor, *Proc. $9^{th}$ Int. Conf. CAV '97*, Haifa, IL,LNCS 1254, pages 12–23. Springer-Verlag, 22–25 Jul. 1997.

[129] J.-P. Queille and J. Sifakis. Verification of concurrent systems in Cesar. In *Proc. Int. Symp. on Programming*, LNCS 137, pages 337–351. Springer-Verlag, 1982.

[130] G. Ramalingam. The undecidability of aliasing. *TOPLAS*, 16(5):1467–1471, Sep. 1994.

[131] F. Ranzato. On the completeness of model checking. In D. Sands, editor, *Proc. $10^{th}$ ESOP '2001*, Genova, IT, 2–6 Apr. 2001, LNCS 2028, pages 137–154. Springer-Verlag, 2001.

[132] A. Rountev and B. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. In R. Wilhelm, editor, *Proc. $10^{th}$ Int. Conf. CC '2001*, Genova, IT, LNCS 2027, pages 20–36. Springer-Verlag, 2–6 Apr. 2001.

[133] J. Rushby. Automated deduction and formal methods. In R. Alur and T.A. Henzinger, editors, *Proc. $8^{th}$ Int. Conf. CAV '96*, number 1102 in LNCS, pages 169–183, New Brunswick, NJ, Jul. /Aug. 1996. Springer-Verlag.

[134] B.G. Ryder, W. Landi, P.A. Stocks, S. Zhang, and R. Altucher. A schema for interprocedural side effect analysis with pointer aliasing. *TOPLAS*, 2001. To appear.

[135] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *$23^{rd}$ POPL*, pages 16–31, St. Petersburg Beach, FL, Jan. 1996. ACM Press.

[136] M. Sagiv, T. Reps, and R. Wilhelm. Shape analysis. In D.A. Watt, editor, *Proc. $9^{th}$ Int. Conf. CC '00*, Berlin, DE, Mar. 25 – Apr. 2, 2000, LNCS 1781, pages 1–17. Springer-Verlag, 2000.

[137] S. Sa di. Model checking guided abstraction and analysis. In J. Palsberg, editor, *Proc. $7^{th}$ Int. Symp. SAS '2000*, Santa Barbara, CA, LNCS 1824, pages 377–396. Springer-Verlag, 29 June – 1 Jul. 2000.

[138] D.A. Schmidt. Data-flow analysis is model checking of abstract interpretations. In *$25^{th}$ POPL*, pages 38–48, San Diego, CA, 19–21Jan. 1998. ACM Press.

[139] J. Schneider and C. Ferdinand. Pipeline behavior prediction for superscalar processors by abstract interpretation. In *Proc. LCTES '99, ACM SIGPLAN Not. 34(7)*, pages 35–44, Atlanta, GE, 5 May 1999, Jul. 1999. ACM Press.

[140] C. Speirs, Z. Somogyi, and H. Søndergaard. Termination analysis for Mercury. In P. Van Hentenryck, editor, *Proc. $4^{th}$ Int. Symp. SAS '97*, Paris, FR, 8–10 Sep. 1997, LNCS 1302, pages 160–171. Springer-Verlag, 1997.

[141] B. Steffen, editor. *Model Checking via Reachability Testing for Timed Automata*, Lisbon, PT, LNCS 1384. Springer-Verlag, 28 Mar. – 4 Apr. 1998.

[142] B. Steffen, editor. *Modular Model Checking of Software*, Lisbon, PT, LNCS 1384. Springer-Verlag, 28 Mar. – 4 Apr. 1998.

[143] H. Theiling and C. Ferdinand. Combining abstract interpretation and ILP for microarchitecture modelling and program path analysis. In *Proc. $19^{th}$ RTSS '98, Madrid, ES*, pages 144–153. IEEE Comp. Soc. Press, 2–4 Dec. 1998. `http://computer.org/proceedings/rtss/9212/9212toc.htm`.

[144] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Syst.*, 18(2–3):157–179, 2000.

[145] A. Venet. Automatic analysis of pointer aliasing for untyped programs. *Sci. Comput. Programming, Special Issue on SAS'96*, 35(1):223–248, Sep. 1999.

[146] S. Verbaeten and D. De Schreye. Termination analysis of tabled logic programs using mode and type information. In A. Middeldorp and T. Sato, editors, *4th FLOPS '99*, Tsukuba, JP, 11–13 Nov. 1999, LNCS 1722, pages 163–178. Springer-Verlag, 1999.

[147] H. Zheng, E. Mercer, and C.J. Myers. Automatic abstraction for verification of timed circuits and systems. In G. Berry, H. Comon, and A. Finkel, editors, *Proc. 30th Int. Conf. CAV '01*, Paris, FR, LNCS 2102, pages 182–193. Springer-Verlag, 16–18 Jul. 2001.