Making Software Verification Tools Really Work[⋆]

Jade Alglave, Alastair F. Donaldson, Daniel Kroening, and Michael Tautschnig

Department of Computer Science, University of Oxford, Oxford, UK

Abstract. We discuss problems and barriers which stand in the way of producing verification tools that are robust, scalable and integrated in the software development cycle. Our analysis is that these barriers span a spectrum from theoretical, through practical and even logistical issues. Theoretical issues are the inherent complexity of program verification and the absence of a common, accepted semantic model in tools. Practical hurdles include the challenges arising from real-world systems features, such as floating-point arithmetic and weak memory. Logistical obstacles we identify are the lack of standard benchmarks to drive tool quality and efficiency, and the difficulty for academic research institutions of allocating resources to tool development. We propose simple measures which we, as a community, could adopt to make the design of serious verification tools easier and more credible. Our long-term vision is for the community to produce tools that are indispensable for a developer but so seamlessly integrated into a development environment, as to be invisible.

1 Introduction

The sophistication and scalability of practical software verification tools has increased dramatically over the last decade. In particular, semi-automatic analysis of moderate-sized software using *model checking* has become viable due to the development of two primary methods: counterexample-guided abstraction refinement (CEGAR) [20] and bounded model checking (BMC) [15]. Counterexample-guided abstraction refinement, realised via predicate abstraction [30] and symbolic model checking [17] of Boolean programs [2], lies at the heart of Microsoft's Static Driver Verifier [4], which is now routinely used by developers of Windows device drivers. Other software model checkers, including BLAST [12], SATABS [22] and CPACHECKER [13] have followed this model and had impact within the research community. Bounded model checking was conceived as a hardware verification technique based on a natural encoding of circuits in propositional logic. Dramatic advances in the performance of SAT and SMT solvers have allowed this technique to be lifted to analyse the behaviour of programs, through a bit-level encoding of variables and operations. Bounded model checking tools such as CBMC [21] and F-SOFT [35] are effective at finding bugs in system-level software, and have been applied in the automotive domain [41]. Recent applications of the k-induction method [43] to software verification [25] have facilitated the use of BMC for verification, not just falsification, of race-freedom properties in software for the Cell BE processor [27].

⋆ Supported by the Engineering and Physical Sciences Research Council (EPSRC) under grants no. EP/G051100/1 and EP/H017585/1.

We believe that a long-term vision for the field is to produce verification tools that are a necessary component of any serious development environment. Despite current success stories, formal verification using model checking based techniques is a long way from such mainstream adoption. Our analysis is that this is due to the following barriers, amongst others:

1. the difficulty of justifying the allocation of resources to tool development in an academic environment;
2. a lack of consensus on what software verification tools should handle, and as a consequence a lack of comparability;
3. a lack of guidance during the software development process, e.g., via unified benchmarks to drive quality and efficiency.

Past articles and discussions have addressed the topic of making verification technology practical, focusing on technology transfer of verification techniques [36] and the role of formal methods in software engineering [44]. The ambitious *Verified Software Initiative* [32] aims to exactly address the problem of practical verification tools, stating as one of its goals "[the construction of] a coherent toolset that automates the theory and scales up to the analysis of industrial-strength software".

In this position paper, we add our voices to the discussion. We believe the above challenges can be addressed via three means: *investment in tools*, encouraged by more stringent requirements for experimental repeatability when submitting verification papers, and a new category of "experimental validation" papers at verification conferences; a *standardisation process* to allow commonality in the way tools are designed and operated; and *challenge benchmarks* to allow tools to be easily tested, improved and compared, thus driving quality.

2 Community Support for Tool Development

Numerous technical problems have to be solved when designing software verification tools, and there is a need for robust tools in order to conduct proper scientific investigation in verification. We briefly discuss ways in which the verification community can support tool development efforts.

2.1 Allocating resources to tool development

A significant portion of research in software verification is carried out by academics, but there are significant barriers for tool development in this community:

It is difficult to obtain research funding for tool development. Proposals for academic funding usually focus on a "big idea" – something novel, seriously challenging, and perhaps a little bit crazy. Without centering on the big idea, a funding proposal will likely be rejected as tame. This sort of blue-sky thinking is important for the development of ground-breaking, non-incremental ideas, but provides no means to develop serious tools over a long period of time. Crucially, resources for long-term tool maintenance and regression testing are usually not requested in a funding proposal.

The priorities of publication venues are a disincentive to building robust tools. Getting a paper accepted to a prestigious venue in automatic verification tends to require a new, deep theoretical idea. While some experimental evaluation is also usually expected, putting together a minimal prototype and concentrating on the theoretical side of a piece of work is a better short-term strategy for getting a paper accepted than painstakingly conducting a rigorous experimental evaluation on a large benchmark set, comparing with a range of other tools. Little or no credit will be given for ensuring that the tool being presented is robust and usable *beyond* the benchmark set used for evaluation. Understandably, time-pressured reviewers tend to scrutinise the theoretical detail of a novel technique readily available in the text of the paper, rather than investing time downloading, installing and experimenting with the associated implementation. They will often not clock whether a reported implementation is a minimal prototype, or a serious piece of software.

In today's environment, two of the main factors used to measure academic success are amount of research money raised, and number of high-quality publications. In this light, the above barriers suggest that an academic who pushes their research group to knock together a series of minimal prototype tools in the run up to major conference deadlines will have greater short-term success than an academic who invests significant time and effort in building robust tools.

2.2 The need for robust tools

One might argue that it is not the responsibility of academics to build robust tools: instead, the job of an academic researcher is to push the boundaries of science by developing novel algorithms, investigating their theoretical properties, and providing proof-of-concept experimental demonstrations. For such proofs-of-concept, aren't minimal prototypes OK? Of course, we are not arguing that academic researchers should be responsible for building industrial-strength tools. But basing research solely on minimal prototypes can be a barrier to scientific progress.

Non-robust tools can lead to vacuous verification. The SLAM verification engine is at the absolute opposite end of the spectrum from being a minimal prototype tool: it is a serious collection of software developed over several years by a dedicated team of researchers, and has led to major uptake of verification technology by industry. This long-term effort has resulted in new insights not possible with the early prototypes: recent work describing the version 2 of the SLAM engine identifies device driver benchmarks where verification using SLAM 2 takes longer than with the original engine, despite a wealth of new optimisations [3]. The reason is that the original version of SLAM is less accurate, and as a result sometimes reported "verification successful" without having established a complete correctness argument. This report on a mature tool suggests that we should be skeptical of experimental results reported for quick prototypes, and illustrates the added value of long-term tool maintenance for research.

Without solid tools, we cannot really do science. Natural sciences hinge upon repeatability of experiments, and the ability to compare complementary or competing techniques in a controlled way. With verification tools, repeatability is often not possible: tools are sometimes to immature to be made available, or snapshots of the versions used to generate experimental results for a given paper are not taken. Tool comparison is also a challenge. Reviewers quite reasonably expect an implementation of a new method to be compared with prior implementations of competing techniques, but it is hard for authors to conduct such a comparison when prior implementations are not available or no longer work.

3 Program Semantics: Minimum Requirements

Designing software verification tools is hindered by the inherent difficulty of the task in general, and the complexity of real-world languages. The inherent difficulty of verification leads to no clear minimum bar for the sorts of input programs that all tools should be capable of handling. The complexity of real-world languages leads to pragmatic decisions related to the handling of semantic features, which are not usually documented and often differ from tool to tool. We now discuss a selection of these issues in some detail.

The specification for a compiler is relatively simple: given semantics for languages A and B, an $A \rightarrow B$ compiler should take any valid program in language A and transform it into a semantically equivalent program in language B. The time taken for transformation should be roughly linear in the size of A. Of course, implementing compilers is challenging, due to the lack of formal specifications for source and target languages, but it is clear that the task of building a compiler is achievable (barring pathological examples [45]).

In contrast, we know from basic undecidability results in computer science that we will never be able to build a verifier that takes an arbitrary program in a Turing-complete language, and decides whether that program is correct (under some appropriate notion of correctness) within some reasonable time bound.

3.1 What should all software verification tools handle, *a minima*?

Because of this inherent difficulty, it is clear that a given software verification tool will not be capable of handling certain input programs. But we would expect that there should be large classes of very simple programs which any respectable software verification tool should be able to cope with. For instance, although loops are hard to analyse in general, a simple program involving loops with a fixed-and-small number of iterations should not be problematic to handle. While pointer-manipulating programs can be tough to analyse, support for straightforward parameter passing by reference via pointers should be non-negotiable. In particular, any verifier should be capable of correctly processing a program with a small and finite state-space (say with fewer than 10,000 states).

In practice, this is often not the case: a prototype verification tool may implement sophisticated algorithms geared towards solving a particular class of problems, but may diverge or crash when invoked on some trivial example program that does not fall within this class. Our viewpoint is that the difficulty of program verification *in general* is not an excuse for tools to perform abysmally, or produce unsound or incomplete results, on simple examples. We need to set the bar somewhere.

3.2 The challenges of semantic features in real-world languages

Research dealing with full-scale languages with complex semantics cannot realistically handle all of their features in all cases. However, it is important that tool designers identify those features which are not handled, and clearly document what the limitations of their tool are. We briefly consider some examples of semantically challenging issues faced by verification tools for C programs:

Bit-level accuracy. Languages in the C family represent numeric types by fixed-width bit-vectors. Thus arithmetic operations may overflow, breaking standard mathematical identities such as $x + 1 > x$. Because arithmetic over/underflow can be the source of subtle bugs, especially in system-level software, it is vital that software verifiers for C-like languages reason with bit-level accuracy. This means parametrising the verifier by a given machine word-size. While early software verifiers tended to use a mathematical model of integers with infinite range, advances in bit-vector solvers led to bit-level accurate tools such as CBMC [21] and F-SOFT [35]. Nowadays, bit-level accurate reasoning is commonplace and widely accepted, primarily owing to the progress modern SAT and SMT solvers have made.

Floating point. Reasoning directly about floating point arithmetic can in principle be achieved by bit-blasting, following the IEEE 754 standard. While this approach is implemented, e.g., by the CBMC tool, it does not scale well due to the immense complexity of floating point circuits. Pragmatic alternatives to supporting floating-point reasoning include treating floating point variables as *fixed* point, or as intervals of *real* numbers, in which case a real arithmetic solver can be exploited. The difficulty with such approaches are that they do not provide accurate results for real programs. While for some users this may be acceptable, for others it may not be, thus such decisions should be clearly documented. Alternative approaches avoid direct reasoning by soundly approximating floating-point computation, either through abstraction [16] or expression canonization [23]. An important open problem is to design fast SMT solvers for floating-point arithmetic [42].

Weak memory models. Analysis tools for *concurrent* programs need to consider the problem of weak memory models exhibited by all modern multicore architectures (e.g., x86 or Power), where the model of computation is not *sequentially consistent* (SC) [38]. Soundness in the presence of weak memory involves considering all possible ways in which memory accesses could be resolved by the hardware, greatly increasing the (already high) complexity of concurrent software analysis. As a result, it is understandable that practical concurrent software verifiers may pragmatically assume an unrealistically strong memory model.

If a tool that aims at handling concurrent programs running on modern multicores supposes SC to be the execution model [26], the tool is strictly unsound, yet perhaps practically useful in finding concurrency bugs or increasing confidence in the correctness of concurrent software. Some programming disciplines such as the *data race free guarantee* (DRF) [1] allow the tool to ignore the details of the memory model, for the discipline enforces the illusion of SC. Hence, tools that assume their input programs to be DRF and SC to be the execution model, e.g., [18], are sound. However, this means that they cannot handle *lock-free synchronisation* [29], a programming style favoured by engineers for its performance, such as for example in the Linux kernel [40].

Again, whether a concurrent software verifier handles weak memory in a sound or restricted manner should be a clearly stated design decision.

System-level features. When applying verification tools to embedded systems software, users typically require support for low-level features such as interrupts, inline assembly and DMA. Correctness for this sort of software often requires careful layout of memory according to machine-specific alignment constraints. These kinds of features are platform-specific, and therefore we do not anticipate a general solution. However, at present, system-level features tend to be modelled in an ad hoc manner for individual applications. A generic framework for describing system-level characteristics relevant to verification, which could then be customised by users for specific needs, does not currently exist and would be be a major step beyond the current state of the art.

Handling source code which is not standard-compliant. While a compiler is typically free to generate arbitrary code where an input program does not conform to the language standard, software verifiers cannot be free to assign specific, arbitrary semantics in such cases. Because bug-finding is an important goal of a software verifier and bugs often arise from lack of adherence to language standards, strictly speaking a verifier should consider *every possible* effect for a statement whose semantics are undefined. Naturally, this strict requirement may not be achievable in practice, and the way to handle or implement certain language features can be controversial. In the absence of a consensus, we believe that developers should state explicitly, and as precisely as possible, how they handle a certain underspecified feature. As an instance, program verifiers may use a fixed order of evaluation of expressions with side effects, while the language standard permits any ordering.

4 The Lack of Guidance in the Process of Writing Tools

Given an input language with well-defined semantics, the question about how to proceed to arrive at a practically useful software verification tool arises. As it is impossible to solve all technical problems in a single step, a suitable form of incremental development must be found. The realization of a research idea, which often involves novel algorithms, effectively results in a conflict of interest: both the novel algorithm must be implemented as efficiently as possible, and several technical hurdles must be overcome. The latter will in parts be well known, whereas other technical challenges might only become visible once the tool has evolved far enough to be applied in verification of real software systems.

To date, we are still in the unfortunate situation that there is only very little preexisting code that offers both high quality and comprehensive documentation to cover all those fundamental technical issues. Such a code base would need to provide a formally defined intermediate representation for at least one major programming language. Furthermore, standard program analyses, as available in compilers, would be expected.

The development of decision procedures, in particular SAT and SMT solvers, has made a lot more progress in the last decade than the development of software verification tools has. Despite continuing evolution and improvement, gritty technical issues tend not to hinder the integration of new algorithms with existing solvers. We identify some of the key advantages in the history of the development of decision procedures, compared with software verifiers:

Publication of technical aspects of decision procedures. Both algorithmic and technical challenges are acknowledged, well documented (in terms of scientific publications – cf. [28,14] for prime examples), and hence technical aspects relevant to tool development are well-understood by the community at large, and easily available to outsiders. Compare this situation to software verification, where entire tools (implemented in hundreds of thousands of lines of code) are often documented in no more than a single conference or journal publication, usually focussing on the tool's core algorithms, presented at a high level of abstraction. It seems that researchers in software verification tools are reluctant to write up technical details, perhaps due to the perception that highly technical papers will be considered engineering, not research, and rejected.

SAT/SMT software architectures have become mature. As a consequence of sharing knowledge about algorithms and problems, in terms of publications and often also source code, developers of decision procedures benefit from lessons learned in other tools [14], hence avoiding redundant re-invention.

Well-defined input languages. Decision procedures benefit from simpler and more formally defined input languages [24,10], compared with software verifiers. The standardisation of programming languages such as C or C++ still leaves many aspects intentionally undefined. A sound software verification tool must thus consider all possible interpretations or offer controllable parameters to the user.

Comparability and competitions. An essential part of any scientific work is a fair and comprehensive comparison to related work. For software verification tools, this is – at present – largely impossible. As such, we are unable to assess progress.

Again, decision procedures have done much better. First, publicly available standard benchmark sets exist to perform comparisons. Second, well established competitions[1][2][3]provide additional incentive to adhere to common input languages, and provide reward for technical improvements. Third, theories and benchmark categories provide precise guidance to users of the technologies, enabling them to select the most suitable tool for their needs [9]. All these measurable facts (performance on standard benchmark sets and supported theories) permit precise, scientific assessment of progress.

We cannot (and should not) change the fact that software verification tools have to deal with complex general-purpose input languages. Yet many such problems could be offloaded to a front end that builds a formally defined intermediate representation. For the reasons laid out in Sec. 2, however, it is challenging for research groups to invest in building such a front end. As an alternative, software verification tool developers could team up to define a subset of a widely used programming language the support of which can be expected from any tool claiming to perform software verification.

Any such standardisation effort will foster comparability; yet two further problems need to be addressed to fully enable comparability: a) for performance comparison, a

[1] `http://www.satcompetition.org/`
[2] `http://www.smtcomp.org/`
[3] `http://www.cs.miami.edu/~tptp/CASC/`

publicly available set of benchmarks in the standardised language must be made available; b) a categorisation similar to theories, as found in SMT, must be established.

5 Proposals for Supporting the Development of Software Verification Tools

Based on the critical assessment of the present situation we propose a way forward. We first discuss possible evolutions in our community, and note the long-term benefits that can be associated with building recognised tools. We then steer towards solutions of technical problems.

5.1 Publication incentives

As discussed in Sec. 2, serious development of tools is not rewarded by the evaluation criteria of publication venues. We propose two strategies for improving this situation:

Repeatability requirements. Publication venues in formal verification should require authors to make implementations of novel algorithms available for inspection and validation by reviewers. To avoid reviewer anonymity being compromised by IP address logging, publication venues should make use of secure means for implementations and benchmarks to be uploaded as part of a paper submission. Such features are already available in submission management systems such as EasyChair. Authors should also provide comprehensive instructions on how to operate the provided software in order to reproduce the paper's results. Reviewers should be encouraged to try to reproduce a selection of results using the provided implementation, and should be encouraged to comment explicitly on whether they have attempted to do so. Review reports should discuss the experience of using the reported implementation, and it should be reasonable to suggest rejecting a paper because the implementation does not work, fails on reasonable examples beyond the benchmark set reported in the paper, or cannot be used due to a lack of comprehensible operating instructions. Working towards standardised interfaces, as proposed in Sec. 5.3, will considerably simplify such evaluations, both for authors and reviewers.

There are two immediate thorny issues associated with such a scheme. First, it relies on reviewers and authors having a common working environment (e.g., using the same operating system and machine word size), and some papers may report experiments on hardware or software which is not universal, or even proprietary. Second, it makes it difficult for industrial practitioners to publish research results where it is not possible to release associated implementations. Reasonable measures would need to be taken to work around these issues, without discouraging valuable contributions from industry. Possible solutions include:

- Requesting that implementations target a specific, widely available OS (virtual machine images may be an option as well).
- Where this is not possible, or where implementations are proprietary, requiring authors to build a web interface through which reviewers can interact with a tool without actually downloading the tool executable.

Also concerned by the problem of experimental repeatability, the databases community has taken exemplary steps to address this at least four years ago [39]. The Call for Papers of the 2008 ACM SIGMOD/PODS Conference includes Experimental Repeatability Requirements in its guidelines for research papers,[4] which are summarised as follows:

> *"To help published papers achieve an impact and stand as reliable referenceable works for future research, the SIGMOD 2008 reviewing process includes an assessment of the extent to which the presented experiments are repeatable by someone with access to all the required hardware, software, and test data. Thus, we attempt to establish that the code developed by the authors exists, runs correctly on well-defined inputs, and performs in a manner compatible with that presented in the paper."*

We strongly believe that the verification community should take a similar stand.

Encouraging experimental validation papers. The databases and systems community also encourages validation of previously published techniques via independent experiments to the extent that it is possible to have a paper accepted by a top databases conference or journal merely by re-implementing and comprehensively evaluating a technique reported previously by a different research group. For example, the Call for Papers of the VLDB 2012 conference includes an *Experiments and Analysis Track*, which "seeks papers that focus on the experimental evaluation of existing algorithms and data structures". This includes explicitly the category of *Result Verification*, for "papers that verify or refute results published in the past and that, through the renewed analysis, help to advance the state of the art".[5]

This publication model allows serious implementation work to be rewarded by prestigious publications, reducing the problem discussed in Sec. 2.1 of implementation work being at odds with short-term goals.

Currently, Calls for Papers at top verification conferences include no such encouragement of result verification, and it is not clear whether a *Result Verification*-style paper would be taken seriously, or rejected due to lack of novelty. We recommend that active steps should be taken to change this situation. The HCI community also discussed the issue of result replication recently via a panel at the 2011 ACM CHI Conference [46].

A note on tool demonstration papers. One might ask at this stage whether tool demonstration papers, which are common in Calls for Papers at verification conferences, serve the goal of encouraging serious implementation. We do not believe this to be the case; tool demonstration papers can often only provide a bite-sized overview of a particular technique.

5.2 Benefits from building tools

The benefits of robust tools to the verification community and beyond are clear but, as discussed in Sec. 2.1, there is little short-term reward for tool development in an

[4] `http://www.sigmod08.org/sigmod_research.shtml`

[5] `http://www.vldb2012.org/call-for-contributions/experiments-and-analysis-track/`

academic environment. However, there are longer-term benefits to serious tool development. We illustrate this by considering three well-known verification tools:

- SPIN, an explicit-state model checker designed at Bell Labs, and now maintained by the NASA/JPL laboratory for reliable software
- SLAM, a CEGAR-based software model checker, designed at Microsoft Research
- PRISM, a probabilistic symbolic model checker designed at the University of Birmingham, and now maintained at the University of Oxford.

Serious software tools can be highly cited. We consider citation counts for the two most highly cited papers on each of these tools:[6]

- SPIN [33,34]: cited 5323 times
- SLAM [6,5]: cited 929 times
- PRISM [37,31]: cited 735 times

These large citation counts indicate significant recognition for the efforts that have gone into development of these tools.

Serious software tools boost research. A robust tool can be used as the basis for a great deal of further research. Looking at the publication records of the key designers of the above tools, we find that each tool has led to tens of further high-quality publications. In the long term, the effort expended in producing a high-quality verification tool pays off, since one does not need to repeatedly construct throw-away prototypes for individual paper deadlines.

5.3 Standards for tool interfaces

In order for a software verification tool to be used (either to reproduce experimental results, or simply to be applied by a practitioner), it is important that the user a) knows how to operate the tool from the command-line or via a GUI, and b) is aware of "magic" keywords and syntactic constructs used in input programs for property specification and/or environment modelling.

To compare two software verification tools geared towards the same input language, one must understand equivalences between command-line or GUI options of both tools, and how equivalent properties and/or environmental assumptions can be specified/modelled using the respective syntactic constructs provided by the tools.

Tool options together with syntax for modelling and specification, together with the language which a verification tool targets, comprise the *interface* of the tool. Clearly the tasks of using an unfamiliar tool and making comparisons between two tools would be eased by *standards* for tool interfaces.

We make the following recommendations in this area:

Focus first on ANSI-C. Given the wide range of programming languages being used today, we cannot expect a long-term convergence on a single language to be supported

[6] Citation counts are taken from Google Scholar on 11 July 2011. For each tool, we have summed the citation count for its key papers. Self-citations have not been excluded.

by all software verification tools. If more front ends for input language processing were available, possibly a convergence towards some intermediate representation could be sought. At present, however, the best we can do is focus on the single programming language most widely supported by the community: ANSI-C. We propose working out a standard interface for ANSI-C verifiers, as discussed below, then using this interface as a basis for tool operation and comparison. If successful, a similar process could be followed for other languages.

Focusing on ANSI-C still requires tools to agree on, or at least identify differences between the precise way in which features that are left undefined in the standard are modelled. We propose a benchmark-based solution to this in Sec. 5.4.

Property specification and environment modelling. In current software verification tools we find a wide range of techniques and assumptions used both in property specification and environment modelling. For example, the BLAST model checker uses the magic variable `__BLAST_NONDET` to specify a nondeterministic value, while with CBMC one obtains a nondeterministic value of type `T` by calling a function declared with return value of type `T` but the body of the function being unavailable. A common verification-level construct is the *assume* statement, which restricts verification to consider only paths on which, when executed, the *assume* statement's guard ϕ evaluates to *true*. Individual tools tend to provide bespoke syntax for *assume* statements (e.g., CBMC uses `__CPROVER_assume(`ϕ`)`). Most verifiers unsoundly but pragmatically agree that the effect of calling a function whose body is unavailable should be a no-op, but that the function should return a nondeterministic result; however, this is rarely a documented feature. Correctness properties may be expressed in some tools via external specification languages [7,11], language extensions [8], or may be embedded in comments using a tool-specific syntax [19].

Given input programs that deviate from the ANSI-C standard, e.g., by reading from invalid memory, or depending on the order in which side-effecting actual parameter expressions are passed, distinct verification tools tend to explore specific behaviours decided upon by the tool implementers. While it is fine for a *compiler* to behave arbitrarily on non-compliant programs, this is not the case for a *verifier*, which should a) detect and report non-compliance, and ideally b) explore all possible ways in which the non-compliant feature could be implemented, in order to catch potential bugs in an implementation-independent manner.

As a result of this variety of approaches, it is currently in general impossible to run the same input program, without modification, through different software verification tools and obtain consistent results. We propose:

- the design of a standard set of syntactic constructs for property specification and environment modelling in C programs. This would be derived from a careful study of the constructs used by existing tools, and possibly also building upon the advances made in other programming languages, such as Spec# [8] or JML [19]. The adoption of such a standard would allow C benchmarks to be evaluated, without modification, by a range of tools.
- the construction of benchmarks to categorise the ways in which verifiers handle non-standard ANSI-C programs. This would allow tool users to understand the

sorts of bugs a verifier will find, and whether two distinct tools will behave similarly when given non-compliant programs.

A standard command-line interface. A much more straightforward proposal is that a standard command-line interface for C verifiers be agreed on. The standard would specify arguments to indicate files to be checked, the main function or functions to be analysed, and possibly even the sorts of generic properties (such as division-by-zero or buffer overflows) to be analysed. We recommend that specification of preprocessor macros, include directories, and other commands shared with compilers, should follow the interface of the widely used GNU C compiler. This standard interface would make it easy to write generic scripts to invoke C verifiers, further easing comparison between tools. Furthermore, the standard would make it possible to build user interfaces in a tool-independent manner, allowing at least simple input programs to be verified with the press of a single button.

5.4 Benchmarks to drive quality, comparability, and competition

The possibility of comparability afforded by a common interface will enable building standard benchmark sets that serve as a basis for fair and scientific comparison. Benchmarks and fair comparison lead towards measurements of progress, and will ultimately enable setting up competitions. We acknowledge that deriving a fair and representative benchmark suite will clearly be even more challenging than, e.g., in case of SAT solvers. Random programs are likely not useful. Different input languages, such as C or Java will require separate benchmark suites. Even if common interfaces are established, different verification tools will remain geared towards different tasks, thus fair comparisons on benchmarks are hard to achieve. We do expect, however, that *branding*, described below, will help to categorise different verification tools according to their strengths.

We expect the design of such a benchmark suite to have several further benefits:

Setting a minimum bar for verification tools. Software model checkers have been available for more than a decade, thus users should reasonably expect them to perform sensibly on small input programs. A set of small benchmark programs will therefore permit to label a given analysis tool a true *software verification tool*. This set of benchmarks will only define *minimal standards*, as discussed in Sec. 3.1. Yet these standard benchmarks will help make tool development easier – one immediately has tests to work towards.

Semantic foundations. A set of benchmarks with precisely defined semantics and independently validated verification results can serve as test whether a tool matches expected semantics for particular language features. As discussed in Sec. 3.2, we acknowledge that not all tools will faithfully handle all semantic aspects of real-world programming languages, and may treat challenging features (e.g., floating-point arithmetic and weak memory) in an unsound but pragmatic way. The proposed benchmark suite will serve as litmus test for verification tools, determining whether a tool faithfully treats a particular language feature, and when this is not the case perhaps even inferring that the tool conforms to a specific known deviation in the way this feature is handled (e.g., determining that fixed-point or real number semantics are used for what should be floating-point reasoning). The benchmarks will allow *branding* of software verification tools, allowing users to quickly get a feeling for whether a tool will be applicable to their particular problem. Our hope is that designers of verification tools will strive for a high-quality branding by the benchmark suite, spurring them on to build robust and usable software.

Driving quality and scalability. If the verification community widely adopts such a benchmark suite, a new verification technique will be taken seriously only if it operates correctly and reasonably efficiently on these benchamrks. This will drive competitiveness, as was observed in the case of decision procedures. If a sufficient level of interface compatibility is achieved, we will be able to run automated competitions, which will provide an incentive to work towards better scalability.

Competition. A competition event with high visibility would foster the transfer of theoretical and conceptual advancements in software verification into practical tools, and would also give credit and benefits to students who spend considerable amounts of time developing verification algorithms and software packages. The first such competition event will compare state-of-the-art software verifiers with respect to effectiveness and efficiency, and the results will be represented at TACAS 2012. [7]

6 Summary

We have discussed the barriers we currently perceive to advancing of the state of the art in software verification tools. We have proposed a number of simple measures which we believe could seriously help this situation: encouragement from the community in the form of a new category of paper and more stringent requirements for experimental reproducibility (inspired by similar measures within the databases and systems community); a common interface for ANSI-C verifiers to enable benchmark compatibility and tool comparison; and a suite of benchmarks which will set a minimum bar for the sophistication of verification tools, provide litmus tests to automatically infer whether and how particular semantic features are handled, and drive the quality and scalability of tools through competitions (inspired by the dramatic competition-driven success in the field of decision procedures).

Technology transfer of hardware verification techniques into practical use proceeded via a sequence of “small steps” [36]. We hope that our proposed measures will act as small steps to continue the transfer of *software* verification techniques into mainstream practice, which has been gaining more and more momentum over the last decade.

Acknowledgments

We thank Vijay D’Silva for stimulating discussion during the writing of this paper, and for useful pointers to related work. We are also grateful to Dragan Bošnački for insightful comments on an earlier draft of this work.

[7] Dirk Beyer, Competition on Software Verification, `http://sv-comp.sosy-lab.org/`

References

1. Adve, S., Hill, M.: Weak ordering – A new definition. In: ISCA (1990)
2. Ball, T., Rajamani, S.: Boolean programs: A model and process for software analysis. Tech. Rep. 2000-14, Microsoft Research (February 2000)
3. Ball, T., Bounimova, E., Kumar, R., Levin, V.: SLAM2: Static driver verification with under 4% false alarms. In: FMCAD. pp. 35–42. IEEE (2010)
4. Ball, T., Bounimova, E., Levin, V., Kumar, R., Lichtenberg, J.: The static driver verifier research platform. In: CAV (2010)
5. Ball, T., Rajamani, S.: The SLAM project: debugging system software via static analysis. In: Principles of Programming Languages (POPL). pp. 1–3. ACM (2002)
6. Ball, T., Rajamani, S.K.: The SLAM toolkit. In: CAV. LNCS, vol. 2102, pp. 260–264. Springer (2001)
7. Ball, T., Rajamani, S.K.: SLIC: a specification language for interface checking (of C). Tech. Rep. MSR-TR-2001-21, Microsoft Research (2001)
8. Barnett, M., DeLine, R., Fähndrich, M., 0002, B.J., Leino, K.R.M., Schulte, W., Venter, H.: The Spec# programming system: Challenges and directions. In: VSTTE. pp. 144–152 (2005)
9. Barrett, C., Stump, A., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB) (2010), `http://www.smt-lib.org`
10. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. In: Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (2010)
11. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: BLAST documentation, `http://www.sosy-lab.org/~dbeyer/blast_doc/`, retrieved 11 July 2011
12. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker BLAST. STTT 9(5-6), 505–525 (2007)
13. Beyer, D., Keremoglu, M.E.: CPACHECKER: A tool for configurable software verification. In: Proc. Computer Aided Verification (CAV). pp. 184–190. LNCS 6806, Springer (2011)
14. Biere, A.: PicoSAT essentials. JSAT 4(2-4), 75–97 (2008)
15. Biere, A., Cimatti, A., Clarke, E., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers (2003)
16. Brillout, A., Kroening, D., Wahl, T.: Mixed abstractions for floating-point arithmetic. In: FMCAD. pp. 69–76. IEEE (2009)
17. Burch, J., Clarke, E., McMillan, K., Dill, D., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. Information and Computation (1992)
18. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. In: POPL. pp. 289–300 (2009)
19. Chalin, P., Kiniry, J.R., Leavens, G.T., Poll, E.: Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In: FMCO. pp. 342–363 (2005)
20. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. Journal of the ACM (JACM) (2003)
21. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (2004)
22. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: Predicate abstraction of ANSI-C programs using SAT. Formal Methods in System Design (FMSD) pp. 105–127 (2004)
23. Collingbourne, P., Cadar, C., Kelly, P.H.J.: Symbolic crosschecking of floating-point and SIMD code. In: EuroSys. ACM (2011)
24. Satisfiability: Suggested Format (1993), `ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/`
25. Donaldson, A., Haller, L., Kroening, D., Rümmer, P.: Software verification using k-induction. In: SAS. LNCS, vol. 6887. Springer (2011)

26. Donaldson, A.F., Kaiser, A., Kroening, D., Wahl, T.: Symmetry-aware predicate abstraction for shared-variable concurrent programs. In: CAV. LNCS, vol. 6806, pp. 356–371. Springer (2011)
27. Donaldson, A.F., Kroening, D., Ruemmer, P.: Automatic analysis of DMA races using model checking and k-induction. Formal Methods in System Design 39(1), 83–113 (2011)
28. Eén, N., Sörensson, N.: An extensible SAT-solver. In: SAT. pp. 502–518 (2003)
29. Fraser, K., Harris, T.: Concurrent programming without locks. ACM Trans. Comput. Syst. 25(2) (2007)
30. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: Computer-Aided Verification (CAV). pp. 72–83. LNCS, Springer (1997)
31. Hinton, A., Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM: A tool for automatic verification of probabilistic systems. In: TACAS. LNCS, vol. 3920, pp. 441–444. Springer (2006)
32. Hoare, C., Misra, J., Leavens, G.T., Shankar, N.: The verified software initiative: A manifesto. ACM Comput. Surv. 41 (October 2009), article 22
33. Holzmann, G.: Design and Validation of Computer Protocols. Prentice Hall (1991)
34. Holzmann, G.J.: The model checker SPIN. IEEE Trans. Software Eng. 23(5), 279–295 (1997)
35. Ivancic, F., Yang, Z., Ganai, M.K., Gupta, A., Shlyakhter, I., Ashar, P.: F-Soft: Software verification platform. In: CAV. LNCS, vol. 3576, pp. 301–306. Springer (2005)
36. Kurshan, R.P.: Verification technology transfer. In: 25 Years of Model Checking. LNCS, vol. 5000, pp. 46–64. Springer (2008)
37. Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM: Probabilistic symbolic model checker. In: Computer Performance Evaluation / TOOLS. LNCS, vol. 2324, pp. 200–204 (2002)
38. Lamport, L.: How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor. IEEE Trans. Comput. 46(7), 779–782 (1979)
39. Manolescu, I., Manegold, S.: Performance evaluation and experimental assessment – conscience or curse of database research? In: VLDB. pp. 1441–1442. ACM (2007)
40. McKenney, P.: `http://www.rdrop.com/users/paulmck/RCU/`
41. Post, H., Sinz, C., Merz, F., Gorges, T., Kropf, T.: Linking functional requirements and software verification. In: RE. pp. 295–302. IEEE Computer Society (2009)
42. Rümmer, P., Wahl, T.: An SMT-LIB theory of binary floating-point arithmetic. In: International Workshop on Satisfiability Modulo Theories (SMT) (2010)
43. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: FMCAD. LNCS, vol. 1954, pp. 108–125. Springer (2000)
44. Steffen, B.: Major threat: From formal methods without tools to tools without formal methods. In: ICECCS. p. 15. IEEE Computer Society (2004), panel discussion
45. Veldhuizen, T.L.: C++ templates are Turing complete. Tech. rep. (2003), `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.3670`, retrieved 11 July 2011
46. Wilson, M.L., Mackay, W., Chi, E., Bernstein, M., Russell, D., Thimbleby, H.: RepliCHI – CHI should be replicating and validating results more: discuss. In: CHI Extended Abstracts. pp. 463–466. ACM (2011)