Vladimir Klebanov
Bernhard Beckert
Armin Biere
Geoff Sutcliffe (Eds.)

COMPARE 2012
Comparative Empirical Evaluation of Reasoning Systems

Proceedings of the International Workshop
June 30, 2012, Manchester, United Kingdom

Editors

Vladimir Klebanov
Karlsruhe Institute of Technology
Institute for Theoretical Informatics
Am Fasanengarten 5, 76131 Karlsruhe, Germany
Email: klebanov@kit.edu

Bernhard Beckert
Karlsruhe Institute of Technology
Institute for Theoretical Informatics
Am Fasanengarten 5, 76131 Karlsruhe, Germany
Email: beckert@kit.edu

Armin Biere
Institute for Formal Models and Verification
Johannes Kepler University
Altenbergerstr. 69, 4040 Linz, Austria
Email: biere@jku.at

Geoff Sutcliffe
Department of Computer Science
University of Miami
P.O. Box 248154, Coral Gables, FL 33124-4245, USA
Email: geoff@cs.miami.edu

Preface

This volume contains the proceedings of the *1st International Workshop on Comparative Empirical Evaluation of Reasoning Systems* (COMPARE 2012), held on June 30th, 2012 in Manchester, UK, in conjunction with the International Joint Conference on Automated Reasoning (IJCAR).

It has become accepted wisdom that regular comparative evaluation of reasoning systems helps to focus research, identify relevant problems, bolster development, and advance the field in general. Benchmark libraries and competitions are two popular approaches to do so. The number of competitions has been rapidly increasing lately. At the moment, we are aware of about a dozen benchmark collections and two dozen competitions for reasoning systems of different kinds. It is time to compare notes.

What are the proper empirical approaches and criteria for effective comparative evaluation of reasoning systems? What are the appropriate hardware and software environments? How to assess usability of reasoning systems, and in particular of systems that are used interactively? How to design, acquire, structure, publish, and use benchmarks and problem collections?

The aim of the workshop was to advance comparative empirical evaluation by bringing together current and future competition organizers and participants, maintainers of benchmark collections, as well as practitioners and the general scientific public interested in the topic.

We wish to sincerely thank all the authors who submitted their work for consideration. All submitted papers were peer-reviewed, and we would like to thank the Program Committee members as well as the additional referees for their great effort and professional work in the review and selection process. Their names are listed on the following pages. We are deeply grateful to our invited speakers—Leonardo de Moura (Microsoft Research) and Cesare Tinelli (University of Iowa)—for accepting the invitation to address the workshop participants. We thank Sarah Grebing for her help in organizing the workshop and compiling this volume.

June 2012

Vladimir Klebanov
Bernhard Beckert
Armin Biere
Geoff Sutcliffe

Program Committee

Bernhard Beckert	Karlsruhe Institute of Technology, Germany
Christoph Benzmüller	Free University Berlin, Germany
Dirk Beyer	University of Passau, Germany
Armin Biere	Johannes Kepler University Linz, Austria
Vinay Chaudhri	SRI International, USA
Koen Claessen	Chalmers Technical University, Sweden
Alberto Griggio	Fondazione Bruno Kessler, Italy
Marieke Huisman	University of Twente, the Netherlands
Radu Iosif	Verimag/CNRS/University of Grenoble, France
Vladimir Klebanov	Karlsruhe Institute of Technology, Germany
Rosemary Monahan	National University of Ireland Maynooth
Michał Moskal	Microsoft Research, USA
Jens Otten	University of Potsdam, Germany
Franck Pommereau	University of Évry, France
Sylvie Putot	CEA-LIST, France
Olivier Roussel	CNRS, France
Albert Rubio	Universitat Politècnica de Catalunya, Spain
Aaron Stump	University of Iowa, USA
Geoff Sutcliffe	University of Miami, USA

Program Co-Chairs

Bernhard Beckert	Karlsruhe Institute of Technology, Germany
Armin Biere	Johannes Kepler University Linz, Austria
Vladimir Klebanov	Karlsruhe Institute of Technology, Germany
Geoff Sutcliffe	University of Miami, USA

Organising Committee

Bernhard Beckert	Karlsruhe Institute of Technology, Germany
Sarah Grebing	Karlsruhe Institute of Technology, Germany
Vladimir Klebanov	Karlsruhe Institute of Technology, Germany

Additional Referees

Sarah Grebing

Table of Contents

Abstracts of Invited Talks

Contributed Papers

Regression Tests and the Inventor's Dilemma

Leonardo de Moura

Microsoft Research, Redmond

Reasoning systems are extensively used at Microsoft. They are used in test case generation, model-based testing, static program analysis, program verification, analysis of firewall policies, program synthesis, geometric problem solving, to cite a few. Reasoning systems are complicated pieces of software, which are very often trying to attack undecidable problems. In this talk, we describe different systems used at Microsoft, how they are evaluated, and challenges upon releasing new versions of successful systems. All systems share the same basic evaluation technique: regression tests. They also face the same challenge: the inventor's dilemma. In all these systems, progress is not monotonic. The latest version of the Z3 theorem prover may enable users to automatically prove many new theorems, provide new features and performance improvements, but it inevitably also fails to prove theorems proved by the previous version. More importantly, we very often have to give up some of our progress to be able to reach the next plateau. We conclude describing how we have been addressing this challenge in the Z3 project.

Introducing StarExec: a Cross-Community Infrastructure for Logic Solving*

Aaron Stump[1], Geoff Sutcliffe[2], and Cesare Tinelli[1]

[1] Department of Computer Science
The University of Iowa
[2] Department of Computer Science
University of Miami

Ongoing breakthroughs in a number of fields depend on continuing advances in the development of high-performance automated reasoning tools, such as SAT solvers, SMT solvers, theorem provers, constraint solvers, rewrite systems, model checkers, and so on. Typically, application problems are translated into (possibly large and complex) formulas for these tools to reason about. Different tradeoffs between linguistic expressiveness and the difficulty of the original problems have led to the adoption of different reasoning approaches and the use of different logics to encode those problems. Solver communities, formed around these different logics, have developed their own research infrastructures to encourage innovation and ease the adoption of their solver technology. Examples include standard formats for the logic problems, libraries of benchmark problems, and solver competitions to spur innovation and further advances. So far, these different infrastructures have been developed separately in the various logic communities, at significant and largely duplicated cost in development effort, equipment and support.

StarExec, currently under development, is a solver execution and benchmark library service aimed at facilitating the experimental evaluation of automated reasoning tools. It will provide a single piece of storage and computing infrastructure to all logic solving communities, reducing the duplication of effort and waste of resources. StarExec will provide a custom web interface and web services running on a cluster of 150-200 compute nodes, with several terabytes of networked disk space. The service will allow community organizers to store, manage and make available benchmark libraries, competition organizers to run competitions, and individual community members to run comparative evaluations of automated reasoning tools on benchmark problems. The StarExec web site will provide facilities to upload and organize benchmarks, browse and query the stored benchmark libraries, view and analyze execution results and statistics, as well as access various data programmatically through a web API.

This talk gives an overview of StarExec, describing its main design, components, functionality, and usage policies, and discusses its current status and development plan.

* Work made possible in large part by the generous support of the National Science Foundation through grants #0958160, 0957438, 1058748, and 1058925.

Evaluating the Usability of Interactive Verification Systems

Bernhard Beckert[1] and Sarah Grebing[2]

[1] Karlsruhe Institute of Technology (KIT)
beckert@kit.edu
[2] University of Koblenz-Landau
sarahgrebing@uni-koblenz.de

Abstract. Usability is an important criterion for measuring and comparing the quality of software systems. It is particularly important for interactive verification systems, which heavily rely on user support to find proofs and that require various complex user interactions.
In this paper, we present a questionnaire for evaluating interactive verification systems based on Green and Petre's Cognitive Dimensions. In a first case study, we have used this questionnaire to evaluate our own tool, the KeY System. The lessons learned from this evaluation relate (1) to the usability of the KeY System and interactive verification systems in general and also (2) gave us insights on how to perform usability evaluations for interactive verification systems.

1 Introduction

Overview For the acceptance, success, and widespread use of software its usability plays a central role. It is an important criterion for measuring and comparing the quality of software systems. And usability is particularly important for interactive verification systems, which heavily rely on user support to find proofs and that require various complex user interactions. However, measuring usability has so far not been the focus of developers of verification systems – instead the community generally concentrates on measuring performance of systems without evaluating to what degree usability effects a system's efficiency.

In general, there are a variety of methods for integrating usability in the development process. This includes analysis methods such as task and user analysis at the planning stage, testing methods where users are monitored while using a system (or a prototype), but also evaluation methods such as questionnaires and interviews [1, 2].

In this paper, we discuss and present a questionnaire for evaluating interactive verification systems based on Green and Petre's Cognitive Dimensions [3]. In a first case study, we have used this questionnaire to evaluate our own tool, the KeY System. The lessons learned from this evaluation relate (1) to the usability of the KeY System and interactive verification systems in general and also (2) to insights on how to perform usability evaluations for interactive verification systems, where such an evaluation can form a basis for improving usability.

Though usability should be taken into account from the beginning of the software life cycle, in this work we concentrate on evaluating the usability of an already existing tool.

Related Work As said above, there is not a lot of work reported on evaluating usability of verification systems or deduction systems in general. A noteworthy exception is Kadoda et al.'s list of desirable features for educational theorem provers [4], which resulted from an evaluation of proof systems based on a questionnaire using Green and Petre's Cognitive Dimensions' framework.

Griffioen and Huisman [5] present a comparison of PVS and Isabelle from a user's perspective. They propose to have something similar to consumer reports for reasoning tools, which can help users choosing the right tool for their application. In this work the two proof tools Isabelle and PVS are compared with respect to their logic, specification language, proof commands, strategies, and the availability of decision procedures, as well as system architecture, proof manager, and user interface. In addition, the user manuals and support tool is taken into account. At the end, the authors give a list of criteria on which the tools have been tested. The lessons we have learned from our evaluation mainly coincide with the aspects given by the authors.

Aitken and Melham analyze errors in interactive proof attempts [6]. They propose the error taxonomy by Zapf et al. as a usability metric. For this the authors have done two user studies with the interactive theorem provers Isabelle and HOL. Experienced users of both systems had to solve a task and the user's interactions were recorded. User errors where then categorized into three types: logical errors, interaction errors, and syntax errors. The authors draw conclusions about the usability of interactive theorem provers. Based on their evaluation, the authors provide some practical advice for the design of proof environments and the user interface of interactive theorem provers.

There are also attempts to improve the user interface of theorem provers, which plays a central role for usability – besides other factors such as the system's response to the user, the design of program logics and specification languages, system documentation etc. For example, Bertot and Théry [7] propose ways to build a user-friendly interface for theorem provers, which include concepts such as "proof-by-pointing", "script management" and "textual explanation of proofs". The user interfaces of particular deduction systems have been evaluated in a number of papers, for example, that of Isabelle [8], that of PVS [9], and that of KIV [10].

There are various competitions (such as CASC, SMT-COMP and SAT), where the performance of automated deduction systems is compared. In recent years, attempts to compare interactive verification systems have changed from comparing just the effectiveness (counting the number of problems that can be solved in arbitrary time) to the comparison of effectiveness and efficiency

(counting how many problems can be solved in limited time). Competitions of that form where held at VSTTE 2010[3] [11], VSTTE 2012[4], and FoVeOOS 2011[5].

Structure of this Paper This paper is structured as follows: Section 2 gives an overview of usability and its evaluation in general. Section 3 briefly describes the KeY System, which we used for our evaluation case study. In Section 4, we present our questionnaire and its design; and in Section 5 we discuss the recruitment of participants for the evaluation. Section 6 contains the results of the evaluation and the lessons learned on the usability of KeY, the usability of verification systems in general, and on the design of the questionnaire. Finally, in Section 7 we draw conclusions and discuss future work.

2 Usability of Software

Software usability is mostly investigated as part of research in the area of human-computer interaction. However, besides a well-designed user interface, other aspects of a system – e.g., good documentation – play an important role for usability.

According to ISO 9241 Part 11 [12], usability is defined as the "extent to which a product can be used by specified users to achieve specified goals with (i) effectiveness, (ii) efficiency, and (iii) satisfaction in a specified context of use." The standard also defines the three terms effectiveness, efficiency, and satisfaction:

Effectiveness: Accuracy and completeness with which users achieve specified goals.

Efficiency: Resources expended in relation to the accuracy and completeness with which users achieve goals

Satisfaction: Freedom from discomfort and positive attitudes towards the use of the product.

Usability is also often defined via five attributes (1) learnability, (2) efficiency, (3) user retention over time, (4) error rate, and (5) satisfaction. Depending on the system, these attributes differ in relevance and may also directly affect each other. For example, high efficiency often leads to reduced learnability (as, for example, key shortcuts need to be learned) [2].

There are standardized questionnaires and evaluation methods for usability that have been widely accepted. One such method is the Software Usability Measurement Inventory (SUMI)[6]. The SUMI questions are statements with which an interviewed user can "agree" or "disagree" (there is also an "undecided" option).

[3] `http://www.macs.hw.ac.uk/vstte10/Competition.html`
[4] `https://sites.google.com/site/vstte2012/compet`
[5] `http://foveoos2011.cost-ic0701.org/verification-competition`
[6] `http://sumi.ucc.ie`

Besides an introductory text about how to answer the statements, the SUMI questionnaire consists of a main part of 50 statements, which have the three possible answers already mentioned, and a smaller part addressing the interviewee's experience level and the importance of the software.

As SUMI is an established method, with a concise questionnaire, it is a low-effort method both for the evaluator and the interviewee. However, there is also a major disadvantage, which is that the feedback consists only of numbers. Therefore no information on how to improve usability in a particular area is gained. The result of evaluating a system with the SUMI method is a "score" reflecting how well the system performed in each of five dimensions (corresponding to the five attributes mentioned above), and in what dimension the system needs improvement.

Another important method in the area of software usability are the cognitive dimensions of notations, first described by Green and Petre [3] and modified into the Cognitive Dimensions framework proposed by Green and Blackwell. It provides a "practical usability tool for everyday analysts and designers" [13]. Rather than being an analytic method, the cognitive dimensions provide a vocabulary to discuss aspects that are cognitively relevant. The concept should of cognitive dimensions allows designers to evaluate their system, to a certain extent, by themselves, without the help of experts [13, 14].

Table 1 (taken from Green's tutorial on cognitive dimensions [13] with modifications from [3]) briefly summarizes the 14 Cognitive Dimensions of Information Artefacts.

3 Evaluation Target: The KeY System

The target for our usability-evaluation case study is the KeY Program Verification System [15, 16] (co-developed by the authors' research group at the Karlsruhe Institute of Technology, Germany and groups at TU Darmstadt, Germany and at Chalmers University, Gothenburg, Sweden).

The target language for verification in the KeY system is Java Card 2.2.1. Java 1.4 programs that respect the limitations of Java Card (no floats, no concurrency, no dynamic class loading) can be verified as well. Specifications are written using the Java Modeling Language (JML).

The program logic of KeY, called Java Card DL, is axiomatised in a *sequent calculus*. Those calculus rules that axiomatise program formulas define a symbolic execution engine for Java Card and so directly reflect the operational semantics. The calculus is written in a small domain-specific language called the *taclet* language [15] that was designed for concise description of rules. Taclets specify not merely the logical content of a rule, but also the context and pragmatics of its application. They can be efficiently compiled not only into the rule engine, but also into the automation heuristics and into the GUI. Depending on the configuration, the axiomatisation of Java Card in the KeY prover uses 1000–1300 taclets.

Table 1. Definition of the cognitive dimensions by Green and Petre [13, 3]

Cognitive Dimension	*Description*
Visibility and Juxtaposability	Visibility: ability to view components easily, respectively is every part of the code simultaneously visible (assuming a large enough display) Juxtaposability: ability to place/view any two components side by side
Error-proneness	Does the design of the notation induce 'careless mistakes'?
Abstraction	An abstraction is a class of entities or a grouping of elements to be treated as one entity, either to lower the viscosity or to make the notation more like the user's conceptual structure
Hidden dependencies	A hidden dependency is a relationship between two components such that one of them is dependent on the other, but that the dependency is not fully visible. In particular, the one-way pointer where A points to B but B does not contain a back-pointer to A
Premature commitment	Constraints on the order of doing things force the user to make a decision before the proper information is available
Secondary notation	Extra information carried by other means than the official syntax
Viscosity	Resistance to change; the cost of making small changes
Closeness of mapping	Closeness of representation to domain
Consistency	Similar semantics are expressed in similar syntactic forms, respectively when some of the language has been learnt, how much of the rest can be inferred?
Diffuseness	Verbosity of language, respectively how many symbols or graphic entities are required to express a meaning?
Hard mental operations	High demand on cognitive resources
Progressive evaluation	Work-to-date can be checked at any time
Provisionality	Degree of commitment to actions or marks
Role expressiveness	The purpose of a component (or an action or a symbol) is readily inferred

The KeY system is not merely a verification condition generator (VCG), but a theorem prover for program logic that combines a variety of automated reasoning techniques with interactive theorem proving. It employs a free-variable sequent calculus for first-order dynamic logic for Java.

While striving for a high degree of automation, the KeY prover features a user interface for presentation of proof states and rule application, aiming at a seamless integration of automated and interactive proving.

The KeY System's user interface consists of a window divided into two parts. On the left side of the window the proof tree is displayed showing the applied rules and case distinctions.

On the right side of the window shows the sequent currently in focus. KeY supports navigation of the proof tree and the application of rules through clicking on (sub-)formulas, and it offers comfortable interaction mechanisms such as drag-and-drop for quantifier instantiations. More on the user interface of the KeY System can be found in [17].

4 Constructing the Questionnaire

Concept of the Questionnaire The main idea behind the design of our questionnaire is to cover the cognitive dimensions [18, 19]. Our idea was to investigate – in general – whether these dimensions can serve as a method for evaluating interactive verification systems and – specifically – in which way the KeY System should be changed to improve its usability.

Table 1 shows the cognitive dimensions [13], and Table 2 shows examples of the corresponding questions from our questionnaire. As the table shows, our questions cover almost all cognitive dimensions except the dimension of *hidden dependencies*.

Note that some dimensions are covered by more than one question. These questions often differ in how specific they are for our chosen target. For example, a question for the dimension of *visibility and juxtaposability* that applies in general to interactive theorem provers is:

> How clear is the arrangement of the formulas in the open goal? Is it possible to determine where a formula results from?.

A question that also relates to *visibility and juxtaposability* but is more specific to the KeY System is:

> To handle formulas to be proven, the KeY System transforms it with normal form rules (e.g., arithmetic rules). Which normal form rules (the ones KeY applies during the automatic verification process) are most annoying or confusing when the automatic process stops and you have to continue interactively?

Specific questions may lead to more specific answers, which may be more useful for improving the system. Some questions we asked even mention suspected problems with usability and ask for a confirmation. On the other hand, such specific questions are often leading and presuppose certain answers. They also make it harder to compare different systems w.r.t. their usability. For that reason our questionnaire includes both general and specific questions. Some questions are half-way between general and specific, such as

> Your automatic proof stops with 1000 closed and 1 open goal. What are the first steps you do?

Table 2. Example questions from the questionnaire (the full version can be found at `http://userpages.uni-koblenz.de/~sarahgrebing/questionnaireForm.pdf`).

Cognitive Dimension	*Questions*
Visibility and Juxtaposability	How clear is the arrangement of the formulas in the open goal? Is it possible to determine where a formula results from?
Error-proneness	Do some kind of mistakes during the interactive verification process seem particularly common or easy to make?
Abstraction	Would you like to have user-defined abstract datatypes?
Premature commitment	Your automatic proof stops with 1000 closed and 1 open goal. What are the first steps you do?
Secondary notation	In JML it is possible to use comments for notes or explanations of the annotation. Would it be useful to have such a feature for proof nodes/subsequents/proof branches in KeY?
Viscosity	If you need to make a change to the previous work (proof, program, or annotation), how easy is it to make the change? Why?
Closeness of mapping	Does the JML notation or the dynamic logic notation allow you to express your intuition why a program is correct with respect to its annotation? Are there cases where the notation is not sufficient?
Consistency	Where there are different parts of the proof/open goal that have a similar meaning, is the similarity clear from the way they appear? Please give examples.
Diffuseness	Does the JML notation or dynamic logic notation (a) let you say what you want reasonably briefly, or is it (b) long-winded? Why?
Hard mental operations	Verifying programs using KeY, what proportion of the time (approximately) are you spending on: quantifier instantiation, finding the right invariant, ...
Progressive evaluation	How easy is it to stop in the middle of creating a proof and check your work so far? Can you find out how much progress you have made?
Provisionality	Other proof systems allow to sketch the proof at a more abstract/higher level (like Isabelle/HOL's tactics). Do you think it could be useful in KeY to sketch proofs if you have an idea how the proof might look like, without giving detailed interactive guidance? If yes, do you have an idea what such a functionality might look like?
Role expressiveness	Would you like to have labels at formulas that indicate the application of which rule the formula resulted from?

Besides instantiating the cognitive dimensions framework, we also included some questions taken from or inspired by the SUMI questionnaire. In addition, there are some questions aimed at gaining information about the interviewees, e.g., their experience level.

The structure of the questionnaire and examples for the questions are described in more detail in the following subsections.

Structure of the Questionnaire Our questionnaire[7] contains 48 questions in total, of which 44 are open questions. It has the following overall structure: after an introductory text describing the evaluation and its goals, we start with some general questions about the interviewee's experience with the KeY System. Next, we ask questions about performing proof tasks and then questions about the proof presentation. The next parts of the questionnaire cover the notation, in our case the JML notation, and the error messages provided by the system. In the last part of the questionnaire, we ask questions about usability in general, about experiences with other proof systems, and about some auxiliary information such as a contact address.

Questions Covering the Cognitive Dimensions Our questions related to cognitive dimensions are mostly instances of the questions in the "cognitive dimensions questionnaire optimized for users" [18, 19]. For example

> When *looking at an open goal*, is it easy to tell what each *sub-sequent* is for in the overall scheme? Why?

is an instance of

> When *reading the notation*, is it easy to tell what each *part* is for in the overall scheme? Why?

where we have instantiated the "notation" with the "open goal" and "part of notation" with "sub-sequent". Note, that such instantiations cannot be done uniformly for all questions. For example, the term "notation" may have to be instantiated with "open goal", "JML notation" or "proof presentation" in different contexts.

According to Kadoda's checklist [4], there are additional dimensions for verification systems, such as assistance (proof plan, next step) and meaningful error messages, which we covered as well. The dimension "assistance" we covered with the question "When the automatic proof process stops, is there enough information given on the screen to continue interactively or do you have to search (e.g., scroll on the screen or click onto the proof tree / search in the proof tree) for the appropriate information?".

SUMI Questions As already recorded, we included a few questions inspired by SUMI besides questions covering the cognitive dimensions. In particular, some of the more general questions resulted from turning the SUMI question into an open question. For example, the SUMI question (question number 2)

[7] There is an online and an offline version of the questionnaire. The offline version can be downloaded at `http://userpages.uni-koblenz.de/~sarahgrebing/questionnaireForm.pdf`.

I would recommend this software to my colleagues.

was turned into the open question

I would recommend the KeY System to people who ...

SUMI also has a few open questions such as "What do you think is the best aspect of this software, and why?" or "What do you think needs most improvement, and why?". The first question we divided into two questions: "List the three most positive or most helpful aspects of the KeY System for the interactive verification process" and "List the three most negative or annoying aspects of KeY concerning the interactive verification process".

Questions Regarding the Interviewee's Experience Level Different user groups with different experience levels have different needs w.r.t. a system's usability. It is therefore important to get information on the interviewee's experience level and how it relates to their answers.

The questions concerning the interviewee's experience with the KeY System included different ratings of the experience level. The interviewees had to choose one of the experience levels (a) little experience, (b) average, (c) above average, and (d) expert (these levels are defined in the questionnaire in more detail). They had to name the largest and most complex project they had verified using the KeY System. They also had to state since when they have been using the KeY System.

5 Recruiting the Participants

To recruit the participants, we asked 25 KeY users either personally or in personalised emails to participate in our evaluation. We got responses from 17 participants. Their experience levels were almost evenly distributed between "average", "above average" and "expert". We had one participant with "little experience". Many (but not all) participants had some relationship with the KeY project. The time they had been using KeY ranged from the early beginnings of the KeY System's development (around 1999) to a few months during a university course.

For a usability evaluation, this is a small sample of interviewees. But this sample gave us some interesting and useful insights both into the usability of KeY and the design of usability questionnaires (as explained in the following sections), and we feel that this is a sufficient sample size for a first evaluation.

For many interactive verification systems, which are developed in academia, recruiting a larger number of participants is difficult. Even more so, if one wants to recruit participants that know different systems and are able to compare them, or if one wants to only recruit participants that have no relation to the institution were the system is being developed (e.g., are enrolled as students) and are thus more likely to give unbiased responses.

For Kadoda's paper [20] about the differences between designers and users of theorem proving assistants, a questionnaire was sent to 27 interviewees, i.e., a

sample size similar to ours. The case study identified the gap between different system views of the designers on the one hand and the users on the other hand. It also highlighted that the cognitive dimensions have an effect on these differences. However, due to the small sample size, Kadoda found it hard to identify particular areas where system designers have to take special care in order to build a usable system.

6 Lessons Learned

6.1 Lessons Learned About Features that are Important for Usability

Many questions we asked in the questionnaire are posed in a way specific to the KeY System. Nevertheless, many of the answers we got and the lessons learned from the survey apply just as well to interactive verification systems in general. In the following these lessons are discussed in more detail.

Proof Presentation: A Major Point for Improvement Almost all participants agreed that the presentation of (partial) proofs – and the formulas and sequences of which proofs consist – is central to the usability of KeY. It is a time-consuming task to inspect the sequences and to reconstruct the structure of the proof by tracing the origin of each subsequence and formula. Thus, a (further) improvement of KeY in this area, which relates to the cognitive dimension of *visibility and juxtaposability*, should take high priority when usability is to be increased.

Documentation: Not Even the Specialists Know Everything Another important area where the usability of KeY could be improved is documentation. About 50% of the participants mentioned a lack of documentation in at least one of their answers. However, it is not one particular part or feature of the system that seems to be in particular need of better documentation, but different participants with different experience voice a wish for different areas where the documentation should be improved: the proof-search strategy settings, the proof rules, the annotation language, and various other system features.

KeY is a rather long-running project and various documentation exists, including a book [15]. This may be the reason why only very few participants asked for documentation in general or for a manual to be written. But the answers show that good documentation is essential. Even highly experienced users and members of the development team asked for certain aspects of the documentation to be improved, which shows that even specialists cannot be expected to know everything about a tool without referring to documentation.

Proof and Change Management: For a Better Overview of What to Prove A good proof and change management contributes to usability as well. This relates to the cognitive dimensions of *viscosity* and *hidden dependencies*. We asked the question of how easy it is to make a change to previous work (proof, program, or annotation). In the KeY System, changing the program to be verified or its

annotation is a simple task. However, if the proofs contain interactive steps, it is time consuming work to redo the proofs which are affected by changes. There is some functionality in KeY for replaying proofs automatically [21, 22], but responses to our questionnaire show that users would like to have more support in this area.

Additional Annotation Mechanisms: A Possibly Helpful Mechanism for Inexperienced Users In the questionnaire we asked about a possible extension of KeY that allows to the addition of comments to nodes in proof trees:

> In JML it is possible to use comments for notes or explanations of the annotation. Would it be useful to have such a feature for proof nodes/subsequents/proof branches in KeY?

This relates to *secondary notation* in the cognitive dimensions. The range of answers to this question shows that users have mixed feelings. The positive answers emphasised that such a feature may be particularly helpful for inexperienced users. Some participants also suggested that proof annotations may be added automatically by the system. That, however, would go beyond the dimension of *secondary notation* and also relate to better *visibility*.

6.2 Lessons Learned About How Users Define Usability of Interactive Verification Systems

We asked the participants what usability of an interactive verification system means for them. The answers we got were manifold, but they mostly supported our assumption that the cognitive dimensions framework is a good model and provides the right concepts for evaluating usability of interactive verification systems.

Areas that were mentioned frequently as being of particular importance for usability are related to proof presentation and the cognitive dimension of *visibility and juxtaposability*. Some typical answers are: "understandable/easy to read presentation of (partial) proofs/open goal, if the verification has to be continued interactively" and "easy location of proof branches".

Another important area is proof guidance as the following answers suggest: "good feedback and guidance when proof attempt fails" and "suggestions on how to proceed".

Documentation of the tool and its rules as well as easy tool usage (with less theoretical background) have been mentioned as features that contribute to the usability for interactive verification systems. The dimension of *viscosity* was also mentioned (e.g., "proof management, what remains to do").

Finally, there were answers relating to the performance of the system and to the user interface in general (e.g., "easy to use interface" and "interaction via mouse and keyboard").

All in all, the answers covered our expectations for features that contribute to the cognitive dimensions and, thus the usability of interactive verification systems:

– proof presentation,
– documentation,
– change management,
– proof guidance,
– feedback mechanism,
– quality of the user interface,
– good performance of the automatic proof search.

6.3 Lessons Learned About How to Evaluate Usability of Interactive Verification Systems

Lessons on How to Pose Questions

Open vs. Closed Questions One important lesson we have learned is that it is not a good idea to have too many open questions as they take more time to answer. The response rate goes down with too many open questions and the answers tend to get shorter and less useful. Also, answers to open questions are harder to evaluate. One should therefore carefully balance open and closed questions and use closed questions where possible.

Clarity of Questions is Important The cognitive dimensions as vocabulary for the usability of a system are a good starting point. But – not surprisingly – one has to carefully think about how to instantiate the concepts. For example, the question

> Where there are different parts of the proof/open goal that have a similar meaning, is the similarity clear from the way they appear? Please give examples.

is an instance of a question from the cognitive dimensions questionnaire. But with this question we gained almost no information because participants did not know how to interpret "similar meaning". The problem seems to be that in the world of formal methods, "meaning" is identified with "semantics". And to a formal-methods person, it is clear that similar semantics may or may not be indicated by appearance (syntax). On the other hand, "meaning" in the above question could also be interpreted as "purpose". So participants got confused.

Another example for a question that was often misunderstood is

> If the performance of the automatic proof process would be improved, would that lead to a better productivity? Or wouldn't it help, because most time is spent elsewhere during the proof process?

More than half of the users interpreted the "performance" as referring to effectiveness, i.e., which problems can be solved at all, and not as referring to efficiency (computation time), which was our intended meaning.

Rating Participants' Experience Level To interpret participant's answers, it is very important to know if there are different classes of users and to what class a participant belongs. A typical example for such classes are users with different experience levels.

We asked the participants to (subjectively) rate their own experience level and we asked for objective measures, namely the size of the biggest project they did with KeY and for how long they have been using KeY.

It turns out that users are not good at judging their experience level. Some who had just used the system in a lab course for two months rated their level to be "average", while some who had been working frequently with the system for years rated themselves as "average" as well. Thus, asking for a self-rating only makes sense if the various levels are well-defined (such as "you are an experienced user if ...").

Improving Usability vs. Measuring Usability One major drawback of the cognitive dimensions is the fact that there is no measure or score for usability with which one can compare different systems. On the other hand, a SUMI evaluation provides a score but does not provide detailed feedback on how to improve usability. So, in follow-up evaluations, we plan to use a better combination of both kinds of questions to get both a score and some feedback on how to improve the system.

Designing Questionnaires for Evaluating Other Systems? It is rather difficult to design a questionnaire that can be used to evaluate arbitrary verification systems. In particular, questions that relate to the cognitive dimensions depend on the system and on the information the systems designers want to gain.

The authors of the "cognitive dimensions questionnaire for users" [18] propose to let interviewees instantiate each question for themselves and let them choose the issues to which they would like to draw attention. This strategy can work, but it requires motivated interviewees that are willing to think hard about what their answers are.

7 Conclusions and Future Work

Our evaluation provided some important insights and lessons both on the usability of the KeY System and interactive verification systems in general and on how to perform usability evaluations for interactive verification systems. Cognitive dimensions have turned out to be a useful basis. And our questionnaire worked very well in giving us the results and feedback we planned for this first evaluation of the KeY System's usability.

As a result, some new features are now being implemented in KeY to improve its usability, in particular a better traceability of formulas and sequences in (partial) proofs. We also investigate how to change the automatic application of simplification rules to improve KeY w.r.t. the cognitive dimensions of *diffuseness* and *hard mental operations*.

Based on our experience, we will improve our questionnaire such that participants can rate the system w.r.t. different dimensions and, thus, provide a measure of usability. In particular, we plan to use that questionnaire on a larger and uniform group of participants, namely students who have used KeY in a lab course.

Acknowledgements

We would like to thank all participants of this evaluation for spending their time to answer the questionnaire. Their help towards improving the usability of the KeY System is greatly appreciated.

References

1. Ferré, X., Juzgado, N.J., Windl, H., Constantine, L.L.: Usability basics for software developers. IEEE Software **18**(1) (2001) 22–29
2. Heinsen, S., ed.: Usability praktisch umsetzen: Handbuch für Software, Web, Mobile Devices und andere interaktive Produkte. Hanser, München (2003)
3. Green, T.R., Petre, M.: Usability analysis of visual programming environments: A 'cognitive dimensions' framework. Journal of Visual Languages and Computing **7** (1996) 131–174
4. Kadoda, G., Stone, R., Diaper, D.: Desirable features of educational theorem provers: A Cognitive Dimensions viewpoint. In: Proceedings of the 11th Annual Workshop of the Psychology of Programming Interest Group. (1996)
5. Griffioen, W.O.D., Huisman, M.: A comparison of PVS and Isabelle/HOL. In Grundy, J., Newey, M.C., eds.: Theorem Proving in Higher Order Logics, 11th International Conference, TPHOLs'98, Canberra, Australia, September 27 - October 1, 1998, Proceedings. LNCS 1479, Springer (1998) 123–142
6. Aitken, J.S., Melham, T.F.: An analysis of errors in interactive proof attempts. Interacting with Computers **12**(6) (2000) 565–586
7. Bertot, Y., Théry, L.: A generic approach to building user interfaces for theorem provers. Journal of Symbolic Computation **25**(2) (1998) 161–194
8. Gast, H.: Engineering the prover interface. In: Proceedings of the User Interfaces for Theorem Provers Workshop (UITP 2010). (2010)
9. Owre, S.: A brief overview of the PVS user interface. In: 8th International Workshop User Interfaces for Theorem Provers (UITP'08), Montreal, Canada (August 2008) Available at `http://www.ags.uni-sb.de/~omega/workshops/UITP08/UITP08-proceedings.pdf`.
10. Haneberg, D., Bäumler, S., Balser, M., Grandy, H., Ortmeier, F., Reif, W., Schellhorn, G., Schmitt, J., Stenzel, K.: The user interface of the KIV verification system: A system description. In: Proceedings of the User Interfaces for Theorem Provers Workshop (UITP 2005). (2005)
11. Klebanov, V., Müller, P., Shankar, N., Leavens, G.T., Wüstholz, V., Alkassar, E., Arthan, R., Bronish, D., Chapman, R., Cohen, E., Hillebrand, M.A., Jacobs, B., Leino, K.R.M., Monahan, R., Piessens, F., Polikarpova, N., Ridge, T., Smans, J., Tobies, S., Tuerk, T., Ulbrich, M., Weiß, B.: The 1st verified software competition: Experience report. In Butler, M., Schulte, W., eds.: FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings. LNCSS 6664, Springer (2011) 154–168

12. ISO: ISO 9241-11:1998 Ergonomic requirements for office work with visual display terminals (VDTs) – Part 11: Guidance on usability. (1998)
13. Green, T.R., Blackwell, A.: Cognitive dimensions of information artefacts: A tutorial. Technical report, BCS HCI Conference (1998) Available at `http://www.cl.cam.ac.uk/~afb21/CognitiveDimensions/CDtutorial.pdf`.
14. Blackwell, A., Green, T.R.: Notational systems – the cognitive dimensions of notations framework. In Carroll, J.M., ed.: HCI Models, Theories, and Frameworks: Toward a Multidisciplinary Science. Interactive Technologies. Morgan Kaufmann, San Francisco, CA, USA (2003) 103–134
15. Beckert, B., Hähnle, R., Schmitt, P.H., eds.: Verification of Object-Oriented Software: The KeY Approach. LNCS 4334. Springer-Verlag (2007)
16. Ahrendt, W., Beckert, B., Hähnle, R., Rümmer, P., Schmitt, P.H.: Verifying object-oriented programs with KeY: A tutorial. In de Boer, F., Bonsangue, M., Graf, S., de Roever, W., eds.: Revised Lectures, 5th International Symposium on Formal Methods for Components and Objects (FMCO 2006), Amsterdam, The Netherlands. LNCS 4709, Springer (2007)
17. Giese, M.: Taclets and the KeY prover. In Aspinall, D., Lüth, C., eds.: Proceedings, User Interfaces for Theorem Provers Workshop, UITP 2003. Volume 103-C of Electronic Notes in Theoretical Computer Science., Elsevier (2004) 67–79
18. Blackwell, A.F., Green, T.R.: A cognitive dimensions questionnaire optimized for users. In: Proceedings of the Twelfth Annual Meeting of the Psychology of Programming Interest Group (PPIG-12). Volume 1-2. (2000) 137–153
19. Blackwell, A., Green, T.R.: A cognitive dimensions questionnaire. Version 5.1.1. At `http://www.cl.cam.ac.uk/~afb21/CognitiveDimensions/CDquestionnaire.pdf` (February 2007)
20. Kadoda, G.: A Cognitive Dimensions view of the differences between designers and users of theorem proving assistants. In: Proceedings of the 12th Annual Workshop on Psychology of Programming (PPIG 12). (2000) Available at `http://ppig.org/papers/12th-kadoda.pdf`.
21. Beckert, B., Klebanov, V.: Proof reuse for deductive program verification. In Cuellar, J., Liu, Z., eds.: Proceedings, Software Engineering and Formal Methods (SEFM), Beijing, China, IEEE Press (2004)
22. Klebanov, V.: Proof reuse. In [15] (2007)

Broadening the Scope of SMT-COMP: the Application Track

Roberto Bruttomesso[1] and Alberto Griggio[2⋆]

[1] Atrenta France
[2] Fondazione Bruno Kessler

Abstract. During the last decade, SMT solvers have seen impressive improvements in performance, features and popularity, and nowadays they are routinely applied as reasoning engines in several domains. The annual SMT solvers competition, SMT-COMP, has been one of the main factors contributing to this success since its first edition in 2005. In order to maintain its significance and positive impact, SMT-COMP needs to evolve to capture the many different novel requirements which applications pose to SMT solvers, which often go beyond a single yes/no answer to a satisfiability check. In this paper, we present a first step in this direction: the "Application" track introduced in SMT-COMP 2011. We present its design and implementation, report and discuss the results of its first edition, and highlight its features and current limitations.

1 Introduction and Motivation

During the last decade, SMT solvers have seen impressive improvements in performance, features and popularity, and nowadays they are routinely applied as reasoning engines in several domains, from verification to planning and scheduling. Part of this success is the result of a standardization process initiated by the introduction of the SMT-LIB 1.2 standard [13] and continued with the currently adopted SMT-LIB 2.0 standard [3].

Before SMT-LIB was introduced, every solver had his own proprietary input language to specify satisfiability queries: the rich variety of SMT approaches in the last decade resulted in the proliferation of different syntaxes, with the effect of complicating experimental comparison between the solvers. It was not uncommon for research groups to maintain, beyond the solver itself, a set of tools for translating between the various languages.

The process of adopting the SMT-LIB standard, however, was not a completely automatic process. At the time of proposing SMT-LIB 1.2 some solvers were already mature tools, usually specialized for a particular task and with a substantial amount of benchmark database in their own language. Certainly switching to a new language at that point was not much in the interest of SMT

⋆ Supported by Provincia Autonoma di Trento and the European Community's FP7/2007-2013 under grant agreement Marie Curie FP7 - PCOFUND-GA-2008-226070 "progetto Trentino", project ADAPTATION.

research groups: changing language implies writing a new parser, maybe new data structures, translate all the benchmark suites, and perform extensive testing. All this effort "without glory" was probably superior to that of maintaining the proprietary language.

The SMT-COMP was a major driving force towards the adoption of the SMT-LIB standard, as it gave the necessary motivation to standardize the input languages. The effort of implementing the new solver infrastructure was justified by the enthusiasm of participating to a friendly race and by the possibility of acquiring visibility in the formal verification and automated deduction communities.

Another big contribution of the SMT-COMP is that of favoring the collection of application-specific benchmarks. Several users of SMT usually submit their benchmarks to the competition initiative as they know that the participating solvers will try their best optimizations to solve them. As a result the submitted benchmarks will be solved in less amount of time and therefore the original application automatically receive a boost in performance.

The Application track. The Application track (as opposed to the traditional Main track) was conceived to stimulate the development of *incremental* SMT solvers: nowadays, SMT solvers are often tightly integrated with other higher-level environments, such as, e.g., a model-checker, from which they receive many successive satisfiability queries to be answered. Although these queries could be solved by just restarting the search from scratch everytime, the solving process is much more efficient if the SMT solver is instructed to cope with them incrementally, by retaining some information from the result of the previous queries: thus, the solver will perform only the new bit of search that is strictly necessary.

Expressing this behavior in a benchmark requires the specification of multiple satisfiability queries, as well as the ability to set and restore backtrack-points in the SMT solver: the SMT-LIB 2.0 standard allows to cope with this requirement by means of the commands `push` and `pop`, which allow to dynamically control the stack of assertions to be solved. As a side note, from the point of view of the SMT-LIB 2.0 the Main track benchmarks can be seen as a restriction of the Application track ones to contain only one query.

2 Design and Implementation

In designing the Application track of the competition, we had to face several different requirements. The first, and perhaps most important, is that we wanted to be able to mimic the interaction between an SMT solver and the higher level application using it as faithfully as possible. In principle, we could have achieved this by selecting some open-source "reference" application from some important domains (e.g. model checking or program analysis), and requiring competitors in the Application track to link their SMT solvers with the selected applications.

Although this solution was initially considered, it was however quickly dropped, for the following reasons. First, this solution puts a lot of burden on the shoul-

```
(declare-fun c0 () Int)
(declare-fun E0 () Bool)
(declare-fun f0 () Bool)
(declare-fun f1 () Bool)
(push 1) ;; push one checkpoint
(assert (and (or (<= c0 (- 3)) (not f1)) (or (not (= c0 0)) (not f0))))
(check-sat)
(pop 1) ;; discard all the formulas asserted after the most recent checkpoint
(declare-fun f2 () Bool)
(declare-fun f3 () Bool)
(declare-fun f4 () Bool)
(declare-fun c1 () Int)
(declare-fun E1 () Bool)
(assert (and (or (>= (+ c1 (* 3 c0)) 0) (not f4)) (or E0 (= c0 c1) (not f2))))
(push 1)
(check-sat)
(assert (and f1 (not f2)))
(check-sat)
(pop 1)
(exit)
```

Fig. 1. Example of a simple trace for the Application track.

ders of potential competitors, thus contributing to the general perception that the barrier for participating in the SMT competition is too high. Second, it makes the competition heavily depend on some specific (versions of) applications, which could result in unwanted bias and/or difficulty in reproducing the results. Finally, this solution is in strong contrast with one of the main goals that SMT-COMP has pursued since its first edition in 2005, which is the promotion of the SMT-LIB standard input format and library of benchmarks for SMT solvers.

A solution that addresses the above points is to generate a *trace* of the interaction between an application and its back-end SMT solver, by exploiting the capabilities of the new SMT-LIB 2.0 language [3] of expressing incremental problems and complex interactions with the SMT solver. This decouples the competition from the applications that provide the benchmarks, it eases the task of reproducing the results, and it allows for collecting, storing and managing benchmarks using the same formats, tools and infrastructure as the main SMT-COMP track. Moreover, it helps in promoting the adoption of the features of the SMT-LIB 2.0 language for specifying incremental SMT problems. In particular, the Application track makes use of the features of the SMT-LIB 2.0 language that allow for specifying a dynamic stack of formulas (by using the `push`, `pop` and `assert` commands) and performing multiple satisfiability checks (via the `check-sat` command) on it. A simple example trace is shown in Figure 1.

A drawback of the latter solution, however, is that solvers can see the whole sequence of queries that occur in the trace before actually solving them. This is in contrast with the "real world" scenario in which applications query the solvers in an interactive, "online" manner, and do not generate the next query until the solver has produced an answer for the current one. In principle, knowing all the queries in advance might allow some solvers to apply some techniques that would not be available in an online setting. To prevent this possibility, we have developed a *trace executor*, a tool which is designed to mimic the interaction

between an application and an SMT solver used as a back-end reasoning engine. More specifically, the trace executor serves the following purposes: (i) it simulates the online interaction by sending single queries to the SMT solver (through their standard input); (ii) it prevents "look-ahead" behaviors of SMT solvers; (iii) it records time and answers for each call, possibly aborting the execution in case of a wrong answer; (iv) it guarantees a fair execution for all solvers by abstracting from any possible crash, misbehavior, etc. that may happen on the application side. The trace executor tool is open source, and it is available from [14]. Its concept and functionalities are similar to those used for the evaluation of different BDD packages for model checking described in [17].

Scoring mechanism. For the first edition of the Application track, the following scoring procedure was implemented. The score for each benchmark is a pair $\langle n, m\rangle$, with $n \in [0, N]$ an integral number of points scored for the benchmark, where N is the number of satisfiability queries in the benchmark. $m \in [0, T]$ is the (real-valued) time in seconds, where T is the timeout. The score of a solver is obtained by summing component-wise the scores for the individual benchmarks. Scores of solvers are compared lexicographically: a solver with a higher n-value wins, with the cumulative time only used to break ties.

The score for a single benchmark is initialized with $\langle 0, 0\rangle$, and then computed as follows. (i) A correctly-reported `sat` or `unsat` answer after s seconds (counting from the previous answer) contributes $\langle 1, s\rangle$ to the score. (ii) An answer of `unknown`, an unexpected answer, a crash, or a memory-out during execution of the query, or a benchmark timeout, aborts the execution of the benchmark and assigns the current value of the score to the benchmark.[3] (iii) The first incorrect answer has the effect of terminating the trace executor, and the returned score for the overall benchmark is $\langle 0, 0\rangle$, effectively canceling the score for the current benchmark. As queries are only presented in order, this scoring system may mean that relatively "easier" queries are hidden behind more difficult ones located at the middle of the query sequence.

Example 1. For example, suppose that there are 3 solvers, $S1$, $S2$ and $S3$, competing on 2 benchmark traces, $T1$ and $T2$, containing respectively 5 and 3 queries, with a timeout of 100 seconds. Suppose that the solvers behave as follows:

- $S1$ solves each of the first four queries of $T1$ in 10 seconds each and the fifth in another 40 seconds. Then, it solves the first 2 queries of $T2$ in 2 seconds each, timing out on the third;
- $S2$ solves the first four queries of $T1$ in 10 seconds each, timing out on the fifth, and then all the 3 queries of $T2$ in 5 seconds each;
- $S3$ solves the first four queries of $T1$ in 2 seconds each, but it incorrectly answers the fifth. It then solves all the 3 queries of $T2$ in 1 second each.

[3] The timeout is set globally for the entire benchmark; there are no individual timeouts for queries.

Then, the scores for the solvers are as follows:

- $S1$ obtains a score of $\langle 5, 80\rangle$ on $T1$, and a score of $\langle 2, 4\rangle$ on $T2$. Its total score is therefore $\langle 7, 82\rangle$;
- $S2$ obtains a score of $\langle 4, 40\rangle$ on $T1$, and a score of $\langle 3, 15\rangle$ on $T2$, resulting in a total of $\langle 7, 55\rangle$;
- $S3$ obtains a score of $\langle 0, 0\rangle$ on $T1$, due to the wrong answer on the last query, and a score of $\langle 3, 3\rangle$ on $T2$. Its total score is therefore $\langle 3, 3\rangle$.

The final ranking of the solvers is therefore $S2$, $S1$, $S3$. ◇

During the discussions following the end of the competition and the preparation for the next edition, some issues were raised concerning the potential bias of the above scoring mechanism towards certain kinds of benchmarks/applications. Benchmarks collected from different applications vary a lot in the number and the difficulty of their individual satisfiability queries. For instance, benchmarks from hardware bounded model checking typically consist of relatively few (in the order of hundreds) incremental SMT calls of increasing size, in which each query might be exponentially more difficult to solve than its predecessor in the sequence. In contrast, benchmarks taken e.g. from software verification based on predicate abstraction consist of hundreds of thousands of satisfiability checks of much more uniform size and complexity. These differences are not properly reflected in the above score, in which each correct answer adds one point to the result, independently of the context/benchmark in which it occurs, and the main ranking criterion is the total number of correct answers, with the execution time used only for breaking ties. As a consequence, solvers that are optimized for benchmarks with many easy queries have a potential advantage over those designed for bounded model checking.

Different solutions for fixing the above problem are currently being evaluated for the 2012 edition of the Application track. In particular, the current candidate proposal is that of giving different weights to queries depending on the benchmark in which they occur. This could be achieved for instance by incrementing the score of $1/N_i$ points for each solved query, where N_i is the total number of queries of the current trace. In this way, solving one more problem in a BMC trace of bound 100 would count much more than solving one more query in a predicate abstraction trace with 100000 trivial satisfiability checks.

3 Benchmarks

The availability of good quality benchmarks is a crucial factor for the success of solver competitions, and the Application track is no exception. In order to ensure the widest possible range of sources and application domains, a public call for benchmarks was issued several months before the competition dates. No restrictions were put on the nature of the benchmarks or the used theories, as long as they conformed to the SMT-LIB 2.0 specification [3] and they represented realistic sequences of incremental calls to an SMT solver issued by a

higher level application. For the first edition of the Application track, more than 6400 benchmarks were submitted, for a total of more than 4800000 satisfiability queries. The following gives brief descriptions of the benchmarks.

BMC and k-Induction queries from the NuSMV Model Checker [7]. These are verification problems on Linear Hybrid Automata and Lustre designs, using linear rational (QF_LRA) and integer (QF_LIA) arithmetic.

BMC and k-Induction queries from the Kratos Software Model Checker [8]. These are verification problems on SystemC designs. Each benchmark comes in two versions: the first using linear rational arithmetic (QF_LRA), and the second using bit-vector arithmetic (QF_BV).

Predicate abstraction queries from the Blast Software Model Checker [4]. The benchmarks have been generated by logging the calls to the Simplify theorem prover made by Blast for computing predicate abstractions of some Windows device driver C programs. They use the combined theory of linear integer arithmetic and uninterpreted functions (QF_UFLIA).

k-Induction queries from the Kind Model Checker [11]. These benchmarks are invariant verification problems on Lustre programs, using the combined theory of linear integer arithmetic and uninterpreted functions (QF_UFLIA).

Access control policy benchmarks from the ASASP project [1]. ASASP implements a symbolic reachability procedure for the analysis of administrative access control policies. The benchmarks use quantifiers, arrays, uninterpreted functions and linear integer arithmetic (AUFLIA).

Selection of Competition Benchmarks. In the main track of SMT-COMP, the subset of the SMT-LIB benchmarks to be used in the competition are selected with an algorithm that ensures a good balance of difficulties, status (`sat` vs `unsat`) and origin of the instances [2]. For the first edition of the Application track, however, in most of the divisions there was no need to perform a selection of benchmarks, given that the number of instances available was sufficiently small that all of them could be included in the competition. The only exceptions were the QF_UFLIA and the AUFLIA divisions, in which the number of benchmarks was too high for including all of them in the competition. In these two cases, we simply picked a subset of the instances with the largest number of incremental queries. As the Application track matures, and more incremental benchmarks become available, we expect to design more sophisticated selection criteria, inspired by those used in the main track.

4 Results

The first edition of the Application track was held during SMT-COMP 2011, as part of the CAV conference. The track was run on the SMT-Exec service [15], using the same infrastructure as the main SMT-COMP.

Participating solvers. Five different solvers took part to the first edition of the Application track. The following gives brief descriptions of the participants.

Boolector 1.4.1 (with SMT-LIB 2.0 parser) [5]. The original BOOLECTOR 1.4.1 was developed by Armin Biere and Robert Brummaryer at the Johannes Kepler University of Linz. BOOLECTOR is one of the most efficient SMT solvers for bit-vectors (QF_BV) and arrays (QF_AUFBV). The participating version was connected with a generic parser for SMT-LIB 2.0 [16], and submitted by the competition organizers as a solver of interest for the SMT community. BOOLECTOR competed in the QF_BV division.

MathSAT 5 [9]. MATHSAT5 is developed by Alberto Griggio, Bas Schaafsma, Alessandro Cimatti and Roberto Sebastiani of Fondazione Bruno Kessler and University of Trento. It is the latest incarnation of a series of solvers with the same name (but independent implementations) that have been developed in Trento as research platforms for SMT since 2002. MATHSAT5 competed in the QF_BV, QF_UFLIA, QF_LIA and QF_LRA divisions.

OpenSMT [6]. OPENSMT is an open-source SMT solver developed by Roberto Bruttomesso, Ondrej Sery, Natasha Sharygina and Aliaksei Tsitovich of Università della Svizzera Italiana, Lugano, with the objective of being an open platform for research, development and detailed documentation on modern SMT techniques. OPENSMT competed in the QF_LRA division.

SMTInterpol [10] 2.0pre. SMTINTERPOL is a solver developed by Jürgen Christ and Jochen Hoenicke of University of Freiburg, with particular focus on proof generation and interpolation. SMTINTERPOL competed in the QF_UFLIA, QF_LRA and QF_LIA divisions.

Z3 3.0 [12]. Z3 3.0 is the latest version of a very popular and efficient SMT solver developed by Leonardo de Moura, Nikolaj Bjørner and Cristoph Wintersteiger at Microsoft Research. Z3 competed in all divisions.

Results. A summary of the results of the 2011 Application track competition is shown in Figure 2. For each division, the table reports the participating solvers, their score expressed as a ratio between the number of solved queries and the total queries (and computed with the procedure described in §2), and the total execution time (not considering the timeouts). No results are reported for the AUFLIA division, since only one solver supporting it (Z3) was submitted.

Discussion. In order to assess the usefulness and significance of the Application track, it is interesting to compare its results with those of the main track of SMT-COMP 2011. Figure 3 summarizes the results of the main track for the solvers participating also in the Application track.[4] For each solver, besides the

[4] For BOOLECTOR, the version used in the main track is newer than those of the Application track.

QF_BV		
Solver	**Score**	**Time**
MathSAT5	1883/2277	14854
Z3	1413/2277	18836
Boolector (+SMT-LIB 2.0)	863/2277	16989

QF_UFLIA		
Solver	**Score**	**Time**
Z3	1238660/1249524	10015
MathSAT5	1237186/1249524	50464
SMTInterpol	1235238/1249524	25440

QF_LRA		
Solver	**Score**	**Time**
MathSAT5	795/1060	3596
Z3	656/1060	10073
SMTInterpol	465/1060	10333
OpenSMT	375/1060	6950

QF_LIA		
Solver	**Score**	**Time**
MathSAT5	12608/13941	40065
Z3	12262/13941	62512
SMTInterpol	9108/13941	66763

Fig. 2. Results of the Application track at SMT-COMP 2011.

score and total execution time, also their absolute ranking in the main track is reported.[5] By comparing the two groups of tables we can see that there are significant differences between the main and the Application tracks in all the divisions. In no single division the rankings are the same across the two tracks, and in three cases out of four the winners are different. There are many possible explanations for such differences, including differences in the nature and the domains of the benchmarks and in the features of the solvers that are put under stress by the two tracks (for example, some solvers apply aggressive and very effective preprocessing techniques, which might not be compatible with incremental problems). Moreover, some divisions in the Application track contain only benchmarks coming from a single source, which might increase the chances of bias towards a particular solver; this was difficult to avoid for the first edition, and it will become less and less evident as the library of incremental benchmarks increases in size. Nevertheless, the fact that there are such visible differences, on benchmarks coming from applications in important domains, is already a sufficient reason to justify the interest in the Application track.

5 Conclusions

In this report we have discussed the motivations, design, implementation and results of the Application track of the SMT-COMP 2011. Our hope is that this new track will contribute in advancing the state-of-the-art of the SMT solvers with respect to their incremental behavior. Such infrastructure is, in fact, very important in recent applications where a tight communication of similar satisfiability queries is demanded. The benchmarks and our implementation of the track were specifically designed to reflect as much as possible this kind of communication.

The results of the first edition of the Application track compared with those of the Main track, shows that the handling incrementality requires different

[5] This is the ranking considering also the other competitors of the main track that did not participate in the Application track, and which are therefore not shown here.

QF_BV			
Solver	**Ranking**	**Score**	**Time**
Z3	1st	188/210	7634
Boolector (v1.5.33)	3rd	183/210	5049
MathSAT5	4th	180/210	6214

QF_UFLIA			
Solver	**Ranking**	**Score**	**Time**
Z3	1st	207/207	205
SMTInterpol	2nd	207/207	2265
MathSAT5	3rd	206/207	2859

QF_LRA			
Solver	**Ranking**	**Score**	**Time**
Z3	1st	195/207	6088
MathSAT5	2nd	193/207	8963
OpenSMT	4th	178/207	18436
SMTInterpol	5th	169/207	16975

QF_LIA			
Solver	**Ranking**	**Score**	**Time**
Z3	1st	203/210	5276
MathSAT5	2nd	190/210	2608
SMTInterpol	3rd	116/210	2917

Fig. 3. Summary of results of the main track of SMT-COMP 2011 for the competitors of the Application track.

optimizations from those used in single-query benchmarks. This motivates us to collect more incremental benchmarks and to increase the visibility of Application track for its extremely practical orientation.

6 Acknowledgements

Morgan Deters was the third organizer of the SMT-COMP 2011. We wish to acknowledge him for having contributed to the definition of rules and scoring mechanisms and for his indispensable work in setting up the computational infrastructure that allowed the competition to be run.

References

1. Alberti, F., Armando, A., Ranise, S.: Efficient symbolic automated analysis of administrative attribute-based RBAC-policies. In: Cheung, B.S.N., Hui, L.C.K., Sandhu, R.S., Wong, D.S. (eds.) Proceedings of ASIACCS. pp. 165–175. ACM (2011)
2. Barrett, C., Deters, M., de Moura, L., Oliveras, A., Stump, A.: 6 Years of SMT-COMP. Journal of Automated Reasoning pp. 1–35 (2012)
3. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0 (December 2010), `http://goedel.cs.uiowa.edu/smtlib/papers/smt-lib-reference-v2.0-r10.12.21.pdf`
4. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast. STTT 9(5-6), 505–525 (2007)
5. Brummayer, R., Biere, A.: Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In: Kowalewski, S., Philippou, A. (eds.) Proceedings of TACAS. LNCS, vol. 5505, pp. 174–177. Springer (2009)
6. Bruttomesso, R., Pek, E., Sharygina, N., Tsitovich, A.: The OpenSMT Solver. In: Esparza, J., Majumdar, R. (eds.) Proceedings of TACAS. LNCS, vol. 6015, pp. 150–153. Springer (2010)

7. Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In: Brinksma, E., Larsen, K.G. (eds.) Proceedings of CAV. LNCS, vol. 2404, pp. 359–364. Springer (2002)
8. Cimatti, A., Griggio, A., Micheli, A., Narasamdya, I., Roveri, M.: Kratos - A Software Model Checker for SystemC. In: Gopalakrishnan, G., Qadeer, S. (eds.) Proceedings of CAV. LNCS, vol. 6806, pp. 310–316. Springer (2011)
9. Griggio, A.: A Practical Approach to Satisability Modulo Linear Integer Arithmetic. JSAT 8(1/2), 1–27 (2012)
10. Hoenicke, J., Christ, J.: SMTInterpol, `http://ultimate.informatik.uni-freiburg.de/smtinterpol/index.html`
11. Kahsai, T., Tinelli, C.: PKind: A parallel k-induction based model checker. In: Barnat, J., Heljanko, K. (eds.) Proceedings of PDMC. EPTCS, vol. 72, pp. 55–62 (2011)
12. de Moura, L.M., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Proceedings of TACAS. LNCS, vol. 4963, pp. 337–340. Springer (2008)
13. Ranise, S., Tinelli, C.: The SMT-LIB Standard: Version 1.2 (2006), `http://goedel.cs.uiowa.edu/smtlib/papers/format-v1.2-r06.08.30.pdf`
14. The SMT-COMP'11 trace executor, `https://es.fbk.eu/people/griggio/smtcomp11/smtcomp11_trace_executor.tar.gz`
15. SMT-Exec, `https://smtexec.org`
16. SMT-LIBv2 parser, `https://es.fbk.eu/people/griggio/misc/smtlib2parser.html`
17. Yang, B., Bryant, R.E., O'Hallaron, D.R., Biere, A., Coudert, O., Janssen, G., Ranjan, R.K., Somenzi, F.: A Performance Study of BDD-Based Model Checking. In: Gopalakrishnan, G., Windley, P.J. (eds.) Proc. of FMCAD. LNCS, vol. 1522, pp. 255–289. Springer (1998)

A Simple Complexity Measurement for Software Verification and Software Testing

- Discussion Paper -

Zheng Cheng*, Rosemary Monahan, and James F. Power

Computer Science Department
National University of Ireland Maynooth
Co. Kildare, Ireland
{zcheng,rosemary,jpower}@cs.nuim.ie

Abstract. In this paper, we used a simple metric (i.e. Lines of Code) to measure the complexity involved in software verification and software testing. The goal is then, to argue for software verification over software testing, and motivate a discussion of how to reduce the complexity involved in software verification. We propose to reduce this complexity by translating the software to a simple intermediate representation which can be verified using an efficient verifier, such as Boogie [2].

Keywords: Intermediate Verification Language, Software Testing, Software Verification, Metrics

1 Introduction

Software testing cannot guarantee that a tested program will be free from errors, whereas software verification can [1]. However, full verification of entire systems is often impossible because of resource limitations and complexity. Therefore, software testing is still the most common technique for ensuring the reliability of software systems [11].

In this paper, we discuss a possible metric for comparing the complexity of software verification and software testing. The goal is to motivate a discussion of how to reduce the complexity involved in software verification, thereby making verification more applicable for industrial usage.

Using different metrics to measure the complexity of software programs is an active research area, see [4] for an overview. We propose using the Lines of Code (LoC) metric to measure the complexity involved in software verification and software testing. The result of LoC-based complexity measurement shows that if using an efficient program verifier, software verification can be no harder than software testing, while improving our confidence in the correctness of our software. We argue that the Boogie verifier is a suitable verifier and suggest how to correctly translate software to a suitable intermediate representation for input to this verifier.

* Funded by John & Pat Hume Scholarship and Doctoral Teaching Scholarship from the Computer Science Department of NUI Maynooth.

Outline A case study is presented in Section 2. The discussion topic of this paper and our view to it are both suggested in Section 3. Finally, the conclusion is made in Section 4.

2 Case Study

As a case study, we demonstrate a two-way sorting algorithm, taken from the software verification competition of VSTTE 2012 [10], and see how many LoC[1] are written to complete its verification task and its testing task[2]. By "complete", we mean that all the pre-defined criteria for each task have been checked thoroughly.

2.1 Criteria Setting

First, we set one criterion for software testing, i.e., functional behaviour (ensuring an array of booleans is sorted in the given order). Then, we manually generate code to meet the criterion. This code is usually referred to as "test cases" and can be executed by a test runner (in this case, the code targets NUnit [9]). We count the LoC of all test cases as the complexity measurement for software testing.

Next, we set three criteria for software verification, i.e. functional behaviour, termination and safety (e.g. ensure no invalid array access). Then, we manually generate annotations (e.g. preconditions, postconditions and loop invariants) to meet the criteria. These annotations can be interpreted by an automated program verifier (in our case we use Dafny [6]). We count the LoC of annotations as the complexity measurement result for software verification. Regarding the conjunction(s) in a proof obligation, the LoC would count the conjunction symbols (e.g. ampersand symbol) plus 1. For example, in Dafny, a postcondition expressed as "requires A & B;", which would count as two LoC. We have to admit that the current methodology for counting LoC of software verification is informal, and requires further research to make it formalized.

2.2 Results

The LoC measurement results for software verification and software testing are listed in Table 1.

Generally, it is infeasible to test the absence of an event [7]. Thus, the termination and safety criteria are more appropriate for verification than testing. For example, if a program executes and does not stop, all that we know is that the program has not halted yet and no conclusion can be derived from such a circumstance. Whereas in program verification, proof of program termination

[1] The LoC is counted by logical line of code, i.e. a statement followed by a domain-specific termination symbol (e.g. semicolon) will count as one logical line of code.

[2] The full description of this case study can be found at:
http://www.cs.nuim.ie/∼zcheng/COMPARE2012/case.html

Question under Study: Two-way Sort Algorithm				
	Functional Behaviour	Termination	Safety	Total
Software Testing	16	N/A	N/A	16
Software Verification	13	0	2	15

Table 1: LoC measurement result for the Two-way Sort Algorithm

can ensure a program will always halt. For example, the Dafny program verifier uses the keyword ***decreases*** to express the variant functions which are used to proof termination. It is also capable of automatically guessing simple variant functions.

Regarding the functional behaviour criterion, we can see the LoC for software testing is greater than for software verification. Moreover, software testing by nature cannot guarantee all the circumstances are tested. Therefore, in order to get more confidence about a program under test, new code (i.e. test cases) is needed. In contrast, the LoC for functional behaviour checking in software verification is a fixed number (i.e. no extra annotations are needed once a program is verified).

One approach for reducing the LoC involved in software verification is using an intermediate verification language such as the Boogie language [2]. For example, the Dafny program verifier translates its program and specifications into the Boogie language, which allows the Dafny program verifier to use the Boogie verifier as its back-end. The Boogie verifier features abstract interpretation for inference of properties such as loop invariants. Moreover, mathematical theories (e.g., set theory and tree theory) are encoded in the Boogie language in advance, which allows Dafny program verifier writing concise model-based specifications. All these features of intermediate verification language can reduce the quantity of annotations that must be discharged in the verification process. Related work shows that program verifiers powered by the Boogie verifier are excellent in accuracy and efficiency [8].

3 How to Reduce the Complexity of Software Verification

We think using a suitable program verifier can lower the complexity of software verification. In [5], we proposed a reliable generic translation framework for the Boogie language (shown in Figure 1), allowing convenient access to the Boogie verifier. The modelling and metamodelling approach [3] provides the foundation of the framework. An intermediate representation, i.e. the Boogie Extension Metamodel, is introduced to bridge the translation from different source languages to the Boogie language, thereby reducing the translation complexity. By the assistance of proposed framework, it is expected that software verification would be accessible for software developers even more.

We also believe that there are many potential solutions to reduce the complexity of software verification, and further discussion on this topic is warranted.

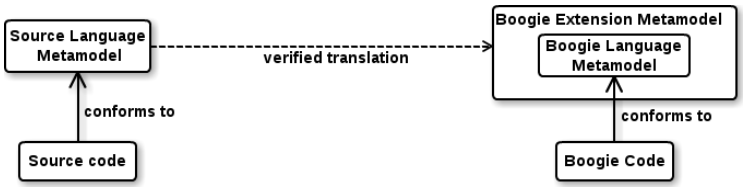

Fig. 1: Overview of Our Proposed Generic Translation Framework

4 Conclusion

In this paper, we used a simple metric (i.e. LoC) to measure the complexity involved in software verification and software testing. The result motivates the use of software verification over software testing, and shows that an efficient program verifier can greatly reduce the verification complexity. How to reduce the complexity of software verification is still an open question that deserves further discussion. In our opinion, the Boogie verifier is a suitable verifier for efficient software verification. To interact with the Boogie verifier, a Boogie program is required as the intermediate representation of the source program to be verified. Our proposed translation framework, based on metamodelling, provides the ideal platform for a reliable translation from a source program to a Boogie program.

References

1. Barnes, J.: High integrity software: The Spark approach to safety and security. Addison-Wesley (2003)
2. Barnett, M., Chang, B.Y.E., Deline, R., Jacob, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. Lecture Notes in Computer Science Vol. 4111 (2006)
3. Bézivin, J.: In search of a basic principle for Model-Driven Engineering. UPGRADE Vol. 5, No. 2 (2004)
4. Cardoso, J., Mendling, J., Neumann, G., Reijers, H.A.: A discourse on complexity of process models (survey paper). Lecture Notes in Computer Science Vol. 4103 (2006)
5. Cheng, Z.: A proposal for a generic translation framework for Boogie language. In: European Conference on Object-Oriented Programming, PhD Workshop (2012)
6. Dafny: http://research.microsoft.com/en-us/projects/dafny/
7. Dijkstra, E.W.: Notes On Structured Programming. Academic Press Ltd. (1972)
8. Klebanov, V., Müller, P., Shankar, N., Leavens, G.T., Wüstholz, V., Alkassar, E., Arthan, R., Bronish, D., Chapman, R., Cohen, E., Hillebrand, M., Jacobs, B., Leino, K.R.M., Monahan, R., Piessens, F., Polikarpova, N., Ridge, T., Smans, J., Tobies, S., Tuerk, T., Ulbrich, M., Weiß, B.: The 1st verified software competition: Experience report. In: Symposium on Formal Methods, Limerick, Ireland (2011)
9. NUnit: http://www.nunit.org/
10. VSTTE.2012.Competition: https://sites.google.com/site/vstte2012/compet/
11. Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.S.: Formal methods: Practice and experience. ACM Computing Surveys Vol. 41, No.4 (2009)

Benchmarking Static Analyzers

Pascal Cuoq, Florent Kirchner, and Boris Yakobowski

CEA LIST*

1 Introduction

Static analysis benchmarks matter. Although benchmarking requires significant effort, it has driven innovation in many areas of Computer Science. Therefore this process and the underlying testcases should be carefully devised. However the problem that static analysis tackles—statically predicting whether a program is correct, or what it does when executed—is so hard that there exist no perfect oracle. For this and other reasons, there is little consensus on desirable and undesirable properties of a static analyzer. This article discusses some of these issues. Its examples involve the minutiae of C, but the principles should generalize to static analysis for most programming languages and, for some, to benchmarks for other partial solutions to undecidable problems.

2 Differing designs, differing goals

Benchmarks organizers should acknowledge that static analyzers use techniques that vary wildly in design and goals. To be representative and useful, the benchmark must be clear on the properties being tested.

At most one issue per test case When static analysis techniques are applied to C, they always only apply to defined executions. This is such a common restriction that it is not usually mentioned by those who make it. As corollary, a testcase should have at most one undefined behavior [4] (if the benchmark is about detecting undefined behaviors) or none (if the benchmark is about detecting another property, say, reachability of a specific label). Having more than one undefined behavior can cause mis-categorizations.

The reason is that it is allowable, and in fact desirable, for static analyzers to use so-called *blocking semantics* where the execution "gets stuck" on undefined behaviors. In fact, all tools use blocking semantics to some extent. In the example below, even high speed/low precision static analyses such as Steengaard's [7] exclude the possibility that `t` coincidentally ends up pointing to `c` because variable `d` happens to follow `c` in memory. Most analyses claim `s` and `t` do not alias. A tool might conclude that `*s == 3`, and that `reachable_p` is not called.

```
int c,  d,  *s = &c,  *t = &d - 1;
*s = 3; *t = 4;
if (*s == 4) reachable_p();
```

* Part of this work has been conducted during the ANR-funded U3CAT project.

Here, it is usual to consider that `reachable_p` is never called, because there are no defined executions that reach the call—although a concrete execution may reach it under circumstances outside the programmer's control. Technically, computing `t = &d - 1` is an undefined behavior [5, §6.5.6:8], and a tool is allowed to consider that execution stops there. Out of practical considerations, some may wait until `t` is dereferenced to assume execution gets stuck. In any case, the correct, consensual answer is that `reachable_p` is unreachable. The tricky situation is when there is no consensus on the semantics of the undefined behavior, as in the next paragraph.

Uninitialized variables as unknown values Some C static analyzers consider uninitialized variables simply as having unknown values. Developers of these tools may even have fallen into the habit of using this behavior as a feature when writing tests. This is their privilege as toolmakers. However, organizers should not assume that all analyzers take this approach. Other analyzers may treat an uninitialized access as the undefined behavior that it is [5, §6.2.4:5, §6.7.8:10, §3.17.2, §6.2.6.1:5] . Using the same "blocking semantics" principle that everyone uses, these analyzers may then consider executions going through the uninitialized access as stuck—and unfairly fail the test.

A better convention to introduce unknown values is a call to a dedicated function. Each toolmaker can then be invited to model the function adequately.

Well-identified goals The SRD test at `http://samate.nist.gov/SRD/view_testcase.php?tID=1737` implies that an analyzer should flag all uses of standard function `realloc`. The justification is that the `realloc`'ed data could remain lying in place in memory if the `realloc` function decided to move the block. However, absent any indication that the data is security-sensitive, the testcase is only measuring the ability of the analyzer to warn for any call to `realloc`: the standard does not specify any circumstances in which `realloc` is guaranteed not to move the pointed block (and having to sanitize the block before calling `realloc` defeats its purpose altogether). The C programming community does not universally consider `realloc` as a dangerous function always to be avoided. Thus an optimal analyzer with 100% marks on this particular problem might be rejected by programmers, who would consider it all noise and no signal.

A solution here is again to establish a simple convention to mark some memory contents as sensitive. If used consistently in all testcases in a benchmark, toolmakers can once and for all describe the relevant adjustments to make their respective tools conform to the convention. In general, the best way to make sure that a benchmark does not misrepresent the strengths and weaknesses of a tool is to include the toolmakers in the process [1].

Bugs in benchmarks We have said that organizers should not *willingly* incorporate undefined behavior in their testcases—unless they are testing the detection of this very defect. At the same time, we recommend organizers embrace the fact

that they will *unwillingly* let a few undesirable bugs slip in. If some of the tested tools are described as warning for undefined behaviors, and if such warnings are emitted for constructs other than the known bug to detect, we recommend that a few of these reports be investigated, just in case.

It is very easy to slip. Canet et al [2] find a bug other than the bug to be detected in the Verisec benchmark [6]. The above-mentioned example 1737 from the NIST SRD contains an undefined behavior: the pointer returned by `realloc` is passed to `printf("...%s\n");` without having been tested for `NULL`. A tool may fail to warn for the `realloc` call being intrinsically dangerous, the debatable property being tested; piling irony upon irony, the same tool may warn about the nearby NULL dereference, a real issue but not the objective of the testcase—and this may be confused as a warning about `realloc`.

Categorizing analyzer outputs Most benchmarks categorize the tools' outputs as "positive" or "negative". This allows results to be synthesized into two well-understood metrics: precision and recall. However, from the point of view of the tool maker, this is frustrating. The differences between techniques are nowhere more visible than in the additional information they can provide. For instance, software model checking can optionally provide a concrete execution path. Abstract interpretation can guarantee that a statement is either unreachable or a run-time error (red alarms in Polyspace). Leaving room for some, but not all, of this information usually reveals the background of the organizers. Finally, since all techniques have their limitations, an "I have detected that the program is outside the scope of what I can do" category would be useful. It's a heartache for toolmakers to have to categorize this situation as either positive, negative or unknown, and it is misleading. It should be kept separate.

3 General benchmarking pitfalls

Do not make the problem too easy Benchmark testcases for static analyzers, in addition to staying clear of undefined behaviors, ought to have unknown inputs. Should they all terminate and come with fixed inputs, then a simple strategy to score on the benchmark is to unroll execution completely [4]. It is nice that a static analyzer is able to do that, but it does not measure how well it fares in actual use, predicting properties for billions of possible inputs at once. As an example of the danger here, the NIST Juliet suite contains testcases with mostly known inputs. A few have unknown inputs, but these are only boolean inputs that are still too easy to brute-force.

Licensing issues The necessity of obtaining a license for proprietary tools provides leverage to vendors: in addition to inquiring about the comparison, they may decline participation, or recommend a particular offering to be tested.

In contrast, the developers of Open Source tools are seldom given these opportunities, leading to two common challenges. One, the distribution of all-inclusive versions—a common behavior in academic communities less subject to

marketing segmentation—require careful configuration, heightening the risk of misuse. This first pitfall can be avoided by contacting makers of Open Source analyzers and allow them to pick options for the task at hand: this is merely the equivalent of picking a tool from an artificially segmented commercial "suite".

Second, some academic licenses for commercial tools include restrictions on publication. One mitigation measure is to inquire whether the restrictions apply when the tool is anonymized [3] and to decide whether to anonymize and publish *before* results are known. Another is to extend the same privileges to all participants in the benchmark; since restrictions can go as far as a veto right on the resulting publication, organizers may well find this option unpalatable. In any case, restrictions that have been applied should be stated in the resulting publication as part of the testing protocol. We do not see quite enough of these caveats, even for comparisons that include commercial tools for which we know that licenses come with such strings attached [8].

4 Conclusion

Several C static analysis benchmarks already exist. It seems timely for this community to follow the same evolution automated proving has, and to move to larger—but still good-natured—competitions. But a large, recognized competition can only emerge if researchers with different backgrounds recognize themselves in it. To this end, basic principles must be agreed on. We propose some in this article. Consensus seems within reach: each of the benchmarks in the short bibliography applies most of the principles we recommend—but none apply them all.

This is a *discussion paper*. The conversation continues with more examples at `http://blog.frama-c.com/index.php?tag/benchmarks`.

Acknowledgments The authors thank Radu Grigore and Éric Goubault for their comments.

References

1. D. Beyer. Competition on software verification - (SV-COMP). In *TACAS*, 2012.
2. G. Canet, P. Cuoq, and B. Monate. A Value Analysis for C Programs. In *SCAM*, 2009.
3. G. Chatzieleftheriou and P. Katsaros. Test-driving static analysis tools in search of C code vulnerabilities. In *COMPSAC*, pages 96–103. IEEE Computer Society, 2011.
4. C. Ellison and G. Roşu. Defining the undefinedness of C. Technical report, University of Illinois, April 2012.
5. International Organization for Standardization. *ISO/IEC 9899:TC3: Programming Languages—C*, 2007. `www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf`.
6. K. Ku, T. E. Hart, M. Chechik, and D. Lie. A buffer overflow benchmark for software model checkers. In *ASE*, pages 389–392. ACM, 2007.
7. B. Steensgaard. Points-to analysis in almost linear time. In *POPL*, pages 32–41. ACM, 1996.
8. M. Zitser, R. Lippmann, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. *SIGSOFT*, 29(6):97–106, October 2004.

The 2nd Verified Software Competition: Experience Report

Jean-Christophe Filliâtre[1,2], Andrei Paskevich[1,2], and Aaron Stump[3]

[1] LRI, Univ Paris-Sud, CNRS, Orsay F-91405
[2] INRIA Saclay-Île-de-France, ProVal, Orsay F-91893
[3] The University of Iowa, Iowa City, Iowa 52242

Abstract. We report on the second verified software competition. It was organized by the three authors on a 48 hours period on November 8–10, 2011. This paper describes the competition, presents the five problems that were proposed to the participants, and gives an overview of the solutions sent by the 29 teams that entered the competition.

1 Introduction

Competitions for high-performance logic-solving tools have played a prominent role in Computational Logic in the past decade or so. Well-established examples include CASC (the competition for first-order provers) [17], the SAT Competition and SAT Race [12], and SMT-COMP (the Satisfiability Modulo Theories Competition) [2]. Many other logic-solving fields have also developed competitions in recent years, with new ones coming online yearly. The reason for this interest is that competitions provide a high-profile opportunity for comparative evaluation of tools. Competitions are good for competitors, who have a chance to increase the visibility of their work by participation in the competition. This is true especially, but not at all exclusively, if they do well. Competitions are good for the field, since they attract attention from a broader audience than might otherwise be following research progress in that area. Finally, competitions are good for potential users of logic-solving tools, who have a chance to see which tools are available and which seem to be doing well on various classes of problems.

Competitions have recently been introduced in the field of deductive software verification. The 1st Verified Software Competition was held in 2010, affiliated with the "Verified Software: Theories Tools and Experiments" (VSTTE) conference [10]. A similar recent event was the COST IC0701 Verification Competition, held in 2011 [5]. These two competitions were quite alike in spirit and organization: The objective was to verify behavioral correctness of several algorithms in a limited amount of time (30–90 minutes per problem), and the participants were free to choose the language of specification and implementation. Another contest of verification tools was the SV-COMP 2012 competition [3], dedicated specifically to fully automated reachability analysis of C programs.

In this paper, we describe the 2nd Verified Software Competition, affiliated with VSTTE 2012. The purposes of this competition were: to help promote approaches and tools, to provide new verification benchmarks, to stimulate further development of verification techniques, and to have fun [1]. In the end, 29 teams competed to solve 5 problems designed by the organizers over a period of three days in early November, 2011. Their solutions, written using a total of 22 different tools, were then judged by the organizers, over the course of the following month. We chose not to make a total ranking list public, but instead identified 6 teams earning gold, silver, or bronze medals. The medalists were announced at VSTTE 2012 (Philadelphia, January 28, 2012) and the 2 gold-medalist teams were each given a 15-minute slot at the conference to present their solution.

The rest of this paper is organized as follows. Section 2 gives an overview of the competition. Then Section 3 describes the five problems and, for each, discusses the solutions that we received. Finally, Section 4 lists the winners and draws some lessons from this competition.

2 Competition Overview

2.1 Schedule

We have chosen to hold the competition over a 48 hours period in order to include more challenging problems than it was feasible for an on-site competition. One source of inspiration for us was the annual ICFP programming contest.

The competition was organized during Fall 2011, with the following schedule:

- Sep 30/Oct 7/Nov 1: the competition is announced on various mailing lists;
- Nov 8, 15:00 UTC: the competition starts (problems are put on the web);
- Nov 10, 15:00 UTC: the competition ends (solutions are sent by email);
- Dec 12: winners are notified privately;
- Jan 28: medalists are announced at VSTTE 2012 and other participants are sent their score and rank privately.

2.2 Technical Organization

The competition website [1] was hosted as a subpage of the VSTTE 2012 website. A Google group was created for the organizers to make announcements, and for the participants to ask questions prior and during the competition and to make their solutions public after the competition if they wish. An email address was created for the participants to submit their solutions.

2.3 Rules

The full description of the competition is available at the competition website [1]. The main rules were the following:

- team work is allowed, but only teams up to 4 members are eligible for the first prize;

Fig. 1. Geographical distribution of the participants.

- any software used in the solutions should be freely available for noncommercial use to the public;
- software must be usable on x86 Linux or Windows;
- participants can modify their tools during the competition.

2.4 Participants

The competition gathered 79 participants, grouped in 29 teams as follows:

- 8 teams of size 1,
- 6 teams of size 2,
- 4 teams of size 3,
- 10 teams of size 4,
- 1 team of size 9.

Participants were from North America, Europe, and Asia, as depicted in Fig. 1. Participants used the following systems, in alphabetic order (the number between parentheses indicates how many teams used that system): ACL2 (1), Agda (3), ATS (1), B (2), BLAST (1), CBMC (1), Coq (7), Dafny (6), Escher (1), Guru (1), HIP (1), Holfoot (1), Isabelle (2), KeY (1), KIV (1), PAT (1), PML (1), PVS (3), Socos (1), VCC (2), VeriFast (1), Ynot (1). 9 teams used at least two different systems.

2.5 Evaluation Process

We approached the rather daunting task of hand-evaluating the solutions as follows. Our goal was to determine the medalists first, and then go back and provide

additional feedback as time permitted to all competitors. So we first triaged the solutions by determining their maximum possible score; that is, maximal points each could possibly earn, based on the parts of the problems it claimed to have solved. For example, only 5 teams claimed to have solved all parts of all the problems. We then heuristically started with the top 8 teams, and each of the organizers closely evaluated their solutions.

The evaluation process was as follows:

1. Read the submission to check that the formal specification conforms to the problem statement.
2. Check the formal proof:
   (a) Run the tool on the input files and check the output (proof found or correctly replayed).
   (b) Manually introduce errors in code or specification (for example, extend loop bounds or weaken a precondition) and rerun the tool to check that input files are not accepted anymore.

As a result of this evaluation, each task was attributed a fraction of its total "worth". We subtracted points based on our assessment that the solution for some part of a problem fell short in some way.

Despite some points being taken away, we were already able to determine our medalists from the first-chosen 8 teams, since no other team had a maximum possible score that could possibly beat the actual score we assessed for the bronze medalists. After we determined the winners, we then divided the remaining solutions among ourselves, to provide written feedback to each team on its solution. Each team was emailed the comments for its solution, once all had been inspected by at least one organizer.

3 Problems and Submissions

Prior to the competition, we prepared 8 problems of various difficulties. Each problem was solved using Why3 [4] and was independently tested by our colleagues Claude Marché and Duckki Oe. Finally, we picked up the following 5 problems:

1. Two-Way Sort (50 points) — sort an array of Boolean values
2. Combinators (100 points) — call-by-value reduction of SK-terms
3. Ring Buffer (150 points) — queue data structure in a circular array
4. Tree Reconstruction (150 points) — build a tree from a list of leaf depths
5. Breadth-First Search (150 points) — shortest path in a directed graph

We selected the problems and assigned the scores according to the time we ourselves and the testers spent devising solutions to them. The three problems we did not include were: Booth's multiplication algorithm, permutation inverse in place [11, p. 176], and counting sort. The former two problems seemed too difficult for a 48-hour competition and the third one was too similar to problems

	2-way sort	SK comb.	ring buffer	tree rec.	BFS
at least one task attempted	25	21	20	28	19
all tasks attempted	20	19	19	17	12
perfect solution	19	10	15	12	9

Fig. 2. Solutions overview (over a total of 29 solutions).

```
two_way_sort(a: array of boolean) :=
  i <- 0;
  j <- length(a) - 1;
  while i <= j do
    if not a[i] then
      i <- i+1
    elseif a[j] then
      j <- j-1
    else
      swap(a, i, j);
      i <- i+1;
      j <- j-1
    endif
  endwhile
```

Fig. 3. Problem 1: Two-Way Sort.

1 and 3. In hindsight, we could have chosen more difficult problems, as the top-ranked participants completed all tasks without much trouble.

Each problem consists of a program to be verified, according to a set of verification tasks ranging from mere safety and termination to full behavioral correctness. A pseudo-code with an imperative flavor is used to describe the programs, but the participants were free to turn them into the language of their choice, including purely applicative languages.

Figure 2 gives an overview of the solutions we received, on a per-problem basis. The remainder of this section describes the problems in detail and provides some feedback on the solutions proposed by the participants.

3.1 Problem 1: Two-Way Sort

The first problem we considered to be easy, and was supposed to be a warm-up exercise for the participants. A pseudo-code to sort an array of Boolean values is given (see Fig. 3). It simply scans the array from left to right with index `i` and from right to left with index `j`, swapping values `a[i]` and `a[j]` when necessary. The verification task are the following:

1. *Safety:* Verify that every array access is made within bounds.
2. *Termination:* Prove that function `two_way_sort` always terminates.
3. *Behavior:* Verify that after execution of function `two_way_sort`, the following properties hold:

$$
\begin{array}{rl}
\textit{terms} & t ::= \mathsf{S} \mid \mathsf{K} \mid (t\ t) \\
\textit{CBV contexts} & C ::= \Box \mid (C\ t) \mid (v\ C) \\
\textit{values} & v ::= \mathsf{K} \mid \mathsf{S} \mid (\mathsf{K}\ v) \mid (\mathsf{S}\ v) \mid ((\mathsf{S}\ v)\ v)
\end{array}
$$

$$
\begin{array}{rl}
\Box[t] = t & C[((\mathsf{K}\ v_1)\ v_2)] \to C[v_1] \\
(C\ t_1)[t] = (C[t]\ t_1) & C[(((\mathsf{S}\ v_1)\ v_2)\ v_3)] \to C[((v_1\ v_3)\ (v_2\ v_3))] \\
(v\ C)[t] = (v\ C[t]) &
\end{array}
$$

Fig. 4. Problem 2: Combinators.

(a) array `a` is sorted in increasing order;
(b) array `a` is a permutation of its initial contents.

Fig. 2 confirms that Problem 1 was the easiest one. Yet we were surprised to see a considerable number of quite laborious solutions, in particular regarding the definition of a permutation (task 3b). We were expecting the participants to pick up such a definition from a standard library, but almost no one did so. Various definitions for the permutation property were used: explicit lists of transpositions together with an interpretation function, bijective mapping of indices, equality of the multisets of elements, etc.

One team used the BLAST model checker to perform tasks 1 (safety) and 2 (termination). Some participants spotted that the loop test `i <= j` can be safely replaced by `i < j` (and proved it).

3.2 Problem 2: Combinators

Problem 2 is slightly different from the other ones. Instead of providing pseudo-code to be verified, this problem simply gives a specification and requires the participants to first implement a function, and then to perform some verification tasks. Namely, the problem defines call-by-value reduction of SK-terms (see Fig. 4) and then proposes one implementation task:

1. Implement a unary function `reduction` which, when given a combinator term t as input, returns a term t' such that $t \to^* t'$ and $t' \not\to$, or loops if there is no such term.

and three verification tasks:

1. Prove that if `reduction`(t) returns t', then $t \to^* t'$ and $t' \not\to$.
2. Prove that `reduction` terminates on any term that does not contain S.
3. Consider the meta-language function ks defined by

$$
\begin{array}{ll}
ks\ 0 & = \mathsf{K}, \\
ks\ (n+1) & = ((ks\ n)\ \mathsf{K}).
\end{array}
$$

   Prove that `reduction` applied to the term $(ks\ n)$ returns K when n is even, and $(\mathsf{K}\ \mathsf{K})$ when n is odd.

One subtlety regarding this problem is that function `reduction` may not terminate. Hence implementing it is already challenging in some systems; that is why implementation is a task in itself. Another consequence is that verification task 1 is a partial correctness result. Tasks 2 and 3 were slightly easier, in particular because it is possible to exhibit a termination argument for `reduction` when applied on S-free terms.

A common error in submissions was forgetting to require a value on the left hand-side of a context ($v\ C$). Indeed, in absence of a dependent type to impose this side condition in the definition of the data type for contexts, one has to resort to an extra well-formedness predicate and it was sometimes accidentally omitted from specifications. Regarding verification task 1, another error was to prove that the returned term was a value without proving that values are irreducible.

One team provided an improved result for verification task 2, showing that any S-free term necessarily reduces to a term of the shape K (K (K . . .)). Another team proved as a bonus that reduction of SII(SII) diverges (with I being defined as SKK).

3.3 Problem 3: Ring Buffer

With problem 3, we are back with traditional imperative programming using arrays. A bounded queue data structure is implemented using a circular array (see Fig. 5) with operations to create a new queue, to clear it, to get or remove the first element, and to add a new element. The verification tasks are the following:

1. *Safety.* Verify that every array access is made within bounds.
2. *Behavior.* Verify the correctness of the implementation w.r.t. the first-in first-out semantics of a queue.
3. *Harness.* The following test harness should be verified.

```
test (x: int, y: int, z: int) :=
  b <- create(2);
  push(b, x);
  push(b, y);
  h <- pop(b); assert h = x;
  push(b, z);
  h <- pop(b); assert h = y;
  h <- pop(b); assert h = z;
```

The challenge of this problem is to come up with a nice specification of the first-in first-out semantics of the queue (task 2). Most participants defined the model of the queue as a list, either as a ghost field in the `ring_buffer` record itself, or as a separate logical function. Some participants, however, opted for algebraic specifications (*i.e.* showing $\texttt{head}(\texttt{push}(b, x)) = x$ and so on). Note that verification task 3 does not require a modular proof using the model defined for the purpose of verification task 2. Thus a mere calculation is accepted as a valid answer for task 3; that solution was used by several participants.

```
type ring_buffer = record
  data : array of int; // buffer contents
  size : int;          // buffer capacity
  first: int;          // queue head, if any
  len  : int;          // queue length
end

create(n: int): ring_buffer :=
  return new ring_buffer(
      data = new array[n] of int;
      size = n; first = 0; len = 0)

clear(b: ring_buffer) :=
  b.len <- 0

head(b: ring_buffer): int :=
  return b.data[b.first]

push(b: ring_buffer, x: int) :=
  b.data[(b.first + b.len) mod b.size] <- x;
  b.len <- b.len + 1

pop(b: ring_buffer): int :=
  r <- b.data[b.first];
  b.first <- (b.first + 1) mod b.size;
  b.len <- b.len - 1;
  return r
```

Fig. 5. Problem 3: Ring Buffer.

This problem appeared to be the second easiest one, judging by the number of perfect solutions, see Fig. 2. Though it was not explicitly required, some solutions are generic w.r.t. the type of elements (and then instantiated on type `int` for task 3). Some participants replaced the `mod` operation by a test and a subtraction, since `b.first + b.len` and `b.first + 1` cannot be greater than 2`b.size` $- 1$. This was accepted.

3.4 Problem 4: Tree Reconstruction

There is a little story behind the fourth problem. It is the last step in Garsia-Wachs algorithm for minimum cost binary trees [7]. It can be stated as follows: given a list l of integers, you have to reconstruct a binary tree, if it exists, such that its leaf depths, when traversed in order, form exactly l. For instance, from the list $1, 3, 3, 2$ one reconstructs the binary tree

```
type tree
Leaf(): tree
Node(l:tree, r:tree): tree

type list
is_empty(s: list): boolean
head(s: list): int
pop(s: list)

build_rec(d: int, s: list): tree :=
  if is_empty(s) then fail; endif
  h <- head(s);
  if h < d then fail; endif
  if h = d then pop(s); return Leaf(); endif
  l <- build_rec(d+1, s);
  r <- build_rec(d+1, s);
  return Node(l, r)

build(s: list): tree :=
  t <- build_rec(0, s);
  if not is_empty(s) then fail; endif
  return t
```

Fig. 6. Problem 4: Tree Reconstruction.

but there is no tree corresponding to the list $1, 3, 2, 2$. A recursive function to perform this reconstruction is given (see Fig. 6; R.E. Tarjan is credited for this code [7, p. 638]). Then the verification tasks are the following:

1. *Soundness.* Verify that whenever function `build` successfully returns a tree, the depths of its leaves are exactly those passed in the argument list.
2. *Completeness.* Verify that whenever function `build` reports failure, there is no tree that corresponds to the argument list.
3. *Termination.* Prove that function `build` always terminates.
4. *Harness.* The following test harness should be verified:
   - Verify that `build` applied to the list $1, 3, 3, 2$ returns the tree `Node(Leaf, Node(Node(Leaf, Leaf), Leaf))`.
   - Verify that `build` applied to the list $1, 3, 2, 2$ reports failure.

One difficulty here is to prove completeness (task 2). Another difficulty is to prove termination (task 3), as it is not obvious to figure out a variant for function `build_rec`. On the contrary, verification task 4 (harness) turns out to be easy as soon as one can execute the code of `build`, as in harness for problem 3.

A delightful bonus from the ACL2 team is worth pointing out: To demonstrate that function `build` is reasonably efficient, they applied it to the LISP code of function `build_rec` itself, as each S-expression can be seen as a binary tree.

```
bfs(source: vertex, dest: vertex): int :=
  V <- {source}; C <- {source}; N <- {};
  d <- 0;
  while C is not empty do
    remove one vertex v from C;
    if v = dest then return d; endif
    for each w in succ(v) do
      if w is not in V then
        add w to V;
        add w to N;
      endif
    endfor
    if C is empty then
      C <- N;
      N <- {};
      d <- d+1;
    endif
  endwhile
  fail "no path"
```

Fig. 7. Problem 5: Breadth-First Search.

3.5 Problem 5: Breadth-First Search

The last problem is a traditional breadth-first search algorithm to find out the shortest path in a directed graph, given a source and a target vertex. The graph is introduced as two abstract data types, respectively for vertices and finite sets of vertices, and a function `succ` to return the successors of a given vertex:

```
type vertex
type vertex_set
succ(v: vertex): vertex_set
```

The code for the breadth-first search is given in Fig. 7. Then the verification tasks are the following:

1. *Soundness.* Verify that whenever function `bfs` returns an integer n this is indeed the length of the shortest path from `source` to `dest`. A partial score is attributed if it is only proved that there exists a path of length n from `source` to `dest`.
2. *Completeness.* Verify that whenever function `bfs` reports failure there is no path from `source` to `dest`.

One difficulty is that the graph is not necessarily finite, thus the code may diverge when there is no path from `source` to `dest`. This was done purposely, to introduce another partial correctness task (as in problem 2). However, some participants asked during the competition if they may assume the graph to be finite, in particular to assume vertices to be the integers $0, 1, \ldots, n-1$, and

sometimes even to use an explicit adjacency matrix for the graph. We answered positively to that request. We also received solutions for this problem that did not rely on this assumption.

Another request was the possibility to rewrite the inner loop of the code (which updates sets `V` and `N`) using set operations (union, difference, etc.). We also agreed. On second thought, the problem would have been nicer if stated this way, that is with the inner loop replaced by the following two assignments:

```
N <- union(N, diff(succ(v), V));
V <- union(V, succ(v));
```

This fifth problem has the lowest number of perfect solutions: 9 out of 29 submissions.

4 Competition Outcome

4.1 And the Winners Are...

A group of 6 excellent submissions with tied scores emerged from our evaluation. Thus we opted for 6 medalists (2 bronze, 2 silver, 2 gold) to avoid discriminating between solutions that were too close. The medalists are:

Gold medal (600 points):

- Jared Davis, Matt Kaufmann, J Strother Moore, and Sol Swords with ACL2 [8,9].
- Gidon Ernst, Gerhard Schellhorn, Kurt Stenzel, and Bogdan Tofan with KIV [16].

Silver medal (595 points):

- K. Rustan M. Leino and Peter Müller with Dafny [13].
- Sam Owre and Natarajan Shankar with PVS [15,14].

Bronze medal (590 points):

- Ernie Cohen and Michał Moskal with VCC [6].
- Jason Koenig and Nadia Polikarpova with Dafny [13].

4.2 Lessons Learned

Evaluation. Probably, the most important conclusion we arrived at during the competition was that evaluation of submitted solutions is a non-trivial and, to some extent, subjective process. This is what sets deductive verification competitions apart from programming contests (where the success can be judged using series of tests) and prover competitions (where a proof trace can be mechanically verified by a trusted certification procedure). In our case, a solution consists of a *formal specification* of the problem and a *program* to solve it, both written in some system-specific language. It is the responsibility of a verification system to check that the program satisfies the requirements posed by the specification, and, to a first approximation, we can trust the verification software to do its job

correctly. Even then a user (and a judge is nothing but an exigent user) must have a good knowledge of the system in question and be aware of the hidden assumptions and turned-off-by-default checks — an issue we stumbled on a couple of times during evaluation.

What is more difficult, error-prone, and time-consuming is checking that a submitted specification corresponds to our intuitive understanding of the problem and its informal description written by humans and for humans. We found no other way to do it than to carefully read the submissions, separating specification from proofs and program code, and evaluating its conformance to our requirements. In this respect, it is not unlike peer-reviewing of scientific publications. Additionally, every system proposes its own language: sometimes quite verbose, sometimes employing a rather exotic syntax, sometimes with specification parts thinly spread among hundreds of lines of auxiliary information. Here we must commend the solutions written for PVS and Dafny for being among the easiest ones to read.

Advice to future organizers. To help improve the evaluation process, we would recommend to organizers of future competitions to gather a kind of "program committee" to which reviewing of submissions could be delegated. Such a committee would benefit from having experts in as many different tools as possible, to help provide more expert reviews of submitted solutions. While a common specification language is out of the question (as the main interest of the contest lies in comparing diverse approaches to formalization), we can strive for more rigid problem descriptions, down to first-order formulations of desired properties. We, as a community, should also push verification system developers towards well-structured and easily readable languages, with a good separation of specification from implementation and proof. Indeed, future organizers might consider requiring the specification of each part of a problem to be clearly indicated, either in comments or even better, by placing the specifications in separate files.

Advice to future competitors. The single biggest issue we had in evaluating solutions was just understanding the specifications of the theorems. One has little choice but to trust the verification tool which claims the solution is correct, but we cannot escape the need to judge whether that solution solves the stated problem or falls short in some way (for example, by adding an assumption that was not explicitly allowed, or by incorrectly formulating some property). So anything a competitor can do to make it as clear and comprehensible as possible what the specification is will help making judging easier and more likely less error-prone. Sometimes even determining which parts of a set of proof scripts or files constitute the specification was challenging. And of course, any special notation or syntax for a tool should be carefully (but briefly) introduced in the README for the submission (and/or possibly inline in the submission itself, where it is first used). Finally, a few submissions we evaluated were not based on plain text files, but rather required a specialized viewer even to look at the solutions. This posed problems for us, particularly when the viewer was only available on one platform (e.g., Windows). We recommend that all tools based on specialized

viewers have some kind of export feature to produce plain text, at least for the statements of theorems proved.

Problem difficulty. When several independent problems are proposed, it is not easy to estimate their relative difficulty in an unbiased manner. The solution that you devise for your problem is not necessarily the best or the simplest one: we were pleasantly surprised to see some participants find more elegant and concise formulations of our algorithms and specifications than those we came up with ourselves (cf. the inner loop in problem 5). Also, what is hard to do in your system of choice might be easy with some other tool. It is always better to draw in several independent and competent testers, preferably using different systems, before the competition.

Verification tasks. An interesting class of verification problems is related to termination issues. Even for systems that admit diverging programs it is not always possible to specify and prove non-termination on a certain input (and we did not include any such task in our problems). Somewhat paradoxically, the systems that are based on logics with total functions (such as Coq) are better suited for this task, as some indirection is required anyway to describe a diverging computation (for example, a supplementary "fuel" parameter).

5 Conclusion

Organizing this competition was a lot of fun — and it seems it was so for the participants as well, which was one of our goals. But it was also a lot of work for us to evaluate the solutions. Obviously this format cannot be kept for future competitions, especially if we anticipate on an even greater number of participants. Alternatives include on-site competitions in limited time (to limit the number of participants), peer-reviewing of the solutions (to limit the workload), and servers with pre-installed verification software (to avoid the installation burden).

Acknowledgments. A large number of people contributed to the success of this competition. We would like to thank: our beta-testers Claude Marché and Duckki Oe; VSTTE 2012 chairs Ernie Cohen, Rajeev Joshi, Peter Müller, and Andreas Podelski; VSTTE 2012 publicity chair Gudmund Grov; LRI's technical staff. Finally, we are grateful to Vladimir Klebanov for encouraging us to write this paper.

References

1. The 2nd Verified Software Competition, 2011. `https://sites.google.com/site/vstte2012/compet`.
2. Clark Barrett, Morgan Deters, Leonardo Moura, Albert Oliveras, and Aaron Stump. 6 years of SMT-COMP. *Journal of Automated Reasoning*, pages 1–35, 2012. to appear in print, available from Springer Online.

3. Dirk Beyer. Competition on Software Verification (SV-COMP). In C. Flanagan and B. König, editors, *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2012*, volume 7214 of *LNCS*, page 504–524. Springer, 2012. Materials available at `http://sv-comp.sosy-lab.org/2012/index.php`.
4. François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond, and Andrei Paskevich. *The Why3 platform*. LRI, CNRS & Univ. Paris-Sud & INRIA Saclay, 2010. `http://why3.lri.fr/`.
5. Thorsten Bormer, Marc Brockschmidt, Dino Distefano, Gidon Ernst, Jean-Christophe Filliâtre, Radu Grigore, Marieke Huisman, Vladimir Klebanov, Claude Marché, Rosemary Monahan, Wojciech Mostowski, Nadia Polikarpova, Christoph Scheben, Gerhard Schellhorn, Bogdan Tofan, Julian Tschannen, and Mattias Ulbrich. The COST IC0701 Verification Competition 2011. In F. Damiani and D. Gurov, editors, *Formal Verification of Object-Oriented Software, Revised Selected Papers Presented at the International Conference, FoVeOOS 2011*. Springer Verlag, 2012. Materials available at `http://foveoos2011.cost-ic0701.org/verification-competition`.
6. Markus Dahlweid, Michal Moskal, Thomas Santen, Stephan Tobies, and Wolfram Schulte. VCC: Contract-based modular verification of concurrent C. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Companion Volume*, pages 429–430, 2009.
7. Adriano M. Garsia and Michelle L. Wachs. A new algorithm for minimum cost binary trees. *SIAM J. on Computing*, 6(4):622–642, 1977.
8. Matt Kaufmann and J. Strother Moore. *ACL2 Version 4.3*. 2011. `http://www.cs.utexas.edu/users/moore/acl2`.
9. Matt Kaufmann, J. Strother Moore, and Panagiotis Manolios. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.
10. Vladimir Klebanov, Peter Müller, Natarajan Shankar, Gary T. Leavens, Valentin Wüstholz, Eyad Alkassar, Rob Arthan, Derek Bronish, Rod Chapman, Ernie Cohen, Mark Hillebrand, Bart Jacobs, K. Rustan M. Leino, Rosemary Monahan, Frank Piessens, Nadia Polikarpova, Tom Ridge, Jan Smans, Stephan Tobies, Thomas Tuerk, Mattias Ulbrich, and Benjamin Weiß. The 1st Verified Software Competition: Experience report. In Michael Butler and Wolfram Schulte, editors, *Proceedings, 17th International Symposium on Formal Methods (FM 2011)*, volume 6664 of *LNCS*. Springer, 2011. Materials available at `www.vscomp.org`.
11. Donald E. Knuth. *The Art of Computer Programming, volume 1 (3rd ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., 1997.
12. D. Le Berre and L Simon, editors. *Special Issue on the SAT 2005 Competitions and Evaluations*, volume 2, 2006.
13. K. Rustan M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In Springer, editor, *LPAR-16*, volume 6355, pages 348–370, 2010.
14. S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.
15. The PVS system. `http://pvs.csl.sri.com/`.
16. W. Reif, G. Schnellhorn, and K. Stenzel. Proving system correctness with KIV 3.0. In William McCune, editor, *14th International Conference on Automated Deduction*, pages 69–72, Townsville, North Queensland, Australia, july 1997.
17. G. Sutcliffe and C. Suttner. The State of CASC. *AI Communications*, 19(1):35–48, 2006.

On the Organisation of Program Verification Competitions

Marieke Huisman[1], Vladimir Klebanov[2], and Rosemary Monahan[3]

[1] University of Twente, The Netherlands
[2] Karlsruhe Institute of Technology, Germany
[3] National University of Ireland Maynooth, Ireland

Abstract. In this paper, we discuss the challenges that have to be addressed when organising program verification competitions. Our focus is on competitions for verification systems where the participants both formalise an informally stated requirement and (typically) provide some guidance for the tool to show it. The paper draws its insights from our experiences with organising a program verification competition at FoVeOOS 2011. We discuss in particular the following aspects: challenge selection, on-site versus online organisation, team composition and judging. We conclude with a list of recommendations for future competition organisers.

1 Introduction

As verification competitions are becoming more popular we are gaining experience on how to organise them. There have been three competitions to-date focusing on a particular form of program verification. In this paper, by program verification, we mean a formal verification process, where a human user contributes in two ways: (a) by formalising an informally stated requirement specification for a program, and (b) by providing (if necessary) some guidance to the verification system to show formally the conformance of the program to the requirement. This setup is due to the strong properties being shown and the heterogeneity of the verification system landscape. It makes such a competition quite different from other competitions in verification (e.g., SV-COMP [1]) or automated reasoning (e.g., CASC [7]), where the formal requirement is identical for all teams and fixed in advance, and no user guidance in showing it is accepted. In our context, organisers have to deal with a whole new range of issues, such as judging the adequacy of the requirement formalisation.

Although verification competitions up until now have varied in their organisation, all of them succeeded in bringing together the verification community to compare their tools and techniques. Hence, it is important that these competitions become a regular event, perhaps co-located with the same conference every year in order to increase participation and build momentum. Through our participation, as both organisers and competitors, we realise that competitions are not mature, and we often make "imperfect" arrangements in order to

increase participation and build momentum. Therefore, such competitions are also a learning process from an organisational viewpoint. In the remainder of this paper we share our experience of organising a verification competition at FoVeOOS 2011. We begin with an overview of verification competitions held so-far. Then, we discuss the challenge selection, on-site versus online organisation, team composition and judging. Finally, we present a list of recommendations for the organisers of future verification competitions.

2 History of Program Verification Competitions

The first program verification competition[4] [5] was an informal event, held during the VSTTE 2010 conference as a prelude to more formal competitions at future meetings. The competition was organised by Natarajan Shankar, SRI International, and Peter Müller, ETH Zürich, who were assisted by Gary Leavens, University of Central Florida, in the judging. The challenges involved simple data types that are supported by most verification tools for sequential or functional programs. The teams, of up to three people, were given five verification exercises with informal specifications, test cases, and pseudo code. The task was to prepare a reproducible verification of executable code relative to a formalisation of the specifications using one or more verification tools. The allotted time was two hours, and the solutions were judged for completeness and elegance as well as correctness.

The second competition was initiated by the COST Action IC0701 [2], whose topic is advancing formal verification of object-oriented software. Organised by Marieke Huisman, University of Twente, Vladimir Klebanov, Karlsruhe Institute of Technology, and Rosemary Monahan, National University of Ireland, Maynooth, the competition aimed to evaluate the usability of verification tools in a relatively controlled experiment that could be easily repeated by others. This competition was inspired by the first (in fact, both Vladimir Klebanov and Rosemary Monahan participated in the first competition), and had a similar format: up to three people forming a team, all participants physically present, and teams using any verification system of their choice. The event took place the afternoon prior to the FoVeOOS 2011 conference. Three challenges were given in natural language and required a solution that consisted of a formal specification and an implementation, where the specification was formally verified with respect to its implementation. In contrast to the VSTTE event, a fixed time slot was assigned for each of the challenges provided. This setup was chosen in order to increase precision of the tool comparisons.

In both of these verification competitions, team registration was not required in advance so participation was quite informal, with student teams especially encouraged. This setup proved to be successful with eleven teams participating in the VSTTE competition and six teams participating in the FoVeOOS competition. It is interesting to note that in each competition every team used

[4] In the sense defined in the introduction.

one verification tool and each tool was represented once. There was no explicit ranking of solutions or a winner announcement. The judging panel manually inspected the solutions and pointed out strengths and weaknesses according to the criteria of completeness, elegance, and automation; these subjective results were presented during the conferences to foster discussions among the participants. In both cases a post-competition paper provided the chance for further discussion and revision of competition solutions.

The third competition [4] had a different format to the previous two: it was an online competition in which participants had 48 hours to attempt five problems that were presented on the conference website. Any programming language, specification language, and verification tool was allowed in the solution. The competition, affiliated with VSTTE 2012 and organised by Jean-Christophe Filliâtre, CNRS, Andrei Paskevich, University of Paris-Sud 11, and Aaron Stump, University of Iowa, attracted 29 teams (79 participants total) using 22 verification tools. Each problem included several sub-tasks, e.g., safety, termination, behavioural correctness, etc., and each sub-task was given a number of points. Submissions from teams of up to four people, were ranked according to the total sum of points they scored. The competition resulted in the award of two gold medals, two silver medals, and two bronze medals. Within each medal class, the teams were tied for points, with the gold medal teams earning perfect scores of 600 points.

3 Challenge Selection

An important step in the organisation of a verification competition is the selection of challenges. Many of the verification challenges posed in competitions so far have been variations of typical "text book" exercises. While posing more open problems is an aspiration, these problems make it difficult to compare solutions and may be daunting to new participants. Here, we discuss the importance of the selection of competition challenges with the aim of making the competitions accessible to all levels of participants, and in particular, making the event attractive to newcomers to the area.

Having a pool of past competition problems in a repository like Verify This [3] assists the challenge selection as one can vary existing problems or can extend the problems to obtain similar or more advanced challenges. On the one hand, such a repository would be a perfect test case for tool developers and a perfect training base for new users. On the other, we want to avoid tool builders tailoring their tools towards the competition database simply to win competitions rather than contributing to the wider verification challenge.

3.1 Challenge Variety

Competition challenges should not (dis)favour a particular tool or approach if at all possible. Verification competitions held so far did not feature tracks or divisions, so quite different tools were pitted against each other. In our opinion,

the challenges issued so far have been favouring tools that target functional programming languages. Tools that target object-oriented languages were in general at a disadvantage.

While introducing tracks or divisions, increases the organisers' effort and requires a bigger participant critical mass, we suggest that the organisers define a set of core challenges, which all teams address, and several "speciality" tracks, where teams can choose the set of challenges that best match the stronger features of their tool. However, it is good to have a nonempty set of core challenges that are attempted by all participants, because one of the important goals of any competition is the comparison of the different solutions.

Additionally, we believe that a good (core) challenge set should be distinctive, i.e., only a few teams should be able to solve each challenge. At the moment, we do not think that there is enough experience with verification competitions, but we believe that eventually the problems should be so distinctive that even strong teams might not be able to solve all challenges (within the given time). Furthermore, we advocate individual time slots for each challenge.

3.2 Challenge Sources

It is important that the challenges are attempted in advance to determine specification pitfalls and to determine the time that should be allocated to solving each challenge. The drawback is that this reduces the pool of people that can participate in a competition.

To obtain a better challenge set with more variety, a possibility is to ask participants to contribute a challenge, with a worked-out solution and an estimate of the required time. This should be something that they consider can be done very well (fast, elegant etc.) with their approach, and would be challenging for other tools. We believe that this will force participants to explicitly consider the strong features of their own tool. It will also help to balance the challenges so that they are not all targeting the same language and problem set. Challenges (and solutions) should be submitted well in advance, to allow the organisers to check that the solution is actually solvable, and does not just use a "trick" that only the tool developers know about.

However it does not seem a good idea to make challenge submission obligatory, as this might prevent non-developer teams from participating, and it would make last minute participation complicated. Instead it would be a better idea to reward challenge submission. For example, bonus points will be awarded if the standard solution is "better" than all submitted solutions.

3.3 Importance of Small Challenges

With certain regularity, we face expectations that competitions should feature larger and more complex challenges. In fact, this has been almost the predominant dimension along which progress of verification as a whole has been evaluated: how large/complex a system can be formally verified? We would like to

argue that this view needs to be supplemented with a different one involving the verification of small, highly controlled challenges.

Two observations lead us to this opinion. First, the larger the challenge, the more difficult/expensive it is to reproduce it. It is a significant advantage if the competition situation can be re-enacted by anybody with access to a verification system and a few hours to dedicate to the task. Larger time demands significantly hamper penetration in the notoriously short-for-time work environment. Second, when working on larger challenges, it is more difficult to keep track of net time and effort spent, as other day-to-day activities (be it sleeping, teaching, or other work) interfere.

It is true that certain tool capabilities that are essential for working on large projects (hierarchical development, abstractions in-the-large, proof and change management) are difficult to test with small challenges. At the same time, a number of larger comparative case studies in formal development and verification have already been carried out. Here we name examples such as the "production cell" case study [6] and the Mondex case study [8].

What is missing is an on-going effort to evaluate *usability* of verification systems, i.e., the amount of work that can be carried out by an average user (preferably not the system's designer) in a fixed amount of time. We conclude that competitions with small focused challenges are an appropriate vehicle for this.

4 On-site versus Online Competitions

To-date, verification competitions have been mainly on-site events, with all team members participating in a common location, for the duration of the competition. These events are typically between two and four hours long with a selection of small challenges to be completed within the allocated timeframe. The location of an on-site competition must provide adequate space for participants, with sufficient caffeine and sugar supplies, and without disturbance from others. It is an advantage to have all teams working in close proximity as this adds to the enthusiasm and adrenaline; with teams reacting when competitors rejoice as a challenge is solved, or lament as the verification doesn't work out as nicely as expected.

From the organisers' perspective the advantage of such a setting is the ease of ensuring that teams participate in accordance with the competition rules (number of teams members involved etc.). A further advantage is that the organisers are available to notice, and clarify, any mis-understandings that arise. From the participants' perspective, the major advantage is the opportunity to interact with users and developers of competing tools after the competition. The momentum, built up through this interaction regarding alternative solution strategies, has led to tool comparisons in a number of conference publications [5] [2]. Another observation is that on-site competition with fixed time slots encourages co-operation between team members. This is due to the urgency of a well-planned solution which solves the challenge in a limited time.

The major disadvantage of an on-site competition is the cost and effort required to get all participants present. Both co-locating the competition with a conference on a related topic so that participants are already on-site and the provision of funding for student team participation have proved to be fruitful strategies.

Online competitions, like those used in many programming competitions, allow for greater participation as teams may participate without travelling. They allow for competitions of longer duration and hence challenges of a larger size. We believe that on-site and online competitions complement each other and should co-exist. For example, larger problems are more suited to off-site challenges that could be issued for tool developers whereas smaller problems are more suited to students/tool users rather than developers.

In either competition setup, we suggest that the interaction between teams after the competition could be increased through the provision of live recordings of the competition (a suggestion, for which we thank Gerhard Schellhorn). Monitoring a team's interaction with a tool could reveal strategies and tips for tool users as well as aiding tool evaluation and increasing interest in verification competitions itself. Of course, the interests of judges and spectators must be balanced with the privacy preferences of participants.

5 Team Composition

With all the verification competitions that have been held so far, one of the dominating questions has been on how to control for the human factor, since it is not meaningful to test the verification system alone. Without proper control, there is a risk that competitions will be dominated by "super experts"—tool developers with many years of experience. They are aware of all the ins-and-outs of the tools, and can even make small changes to the tool during the competition. They also know how to tweak the specifications so that they are easily expressed in the tool's input language and are easily accepted by the tool.

Several ideas exist on how to ensure that super expert users do not skew the competition to their advantage: one could allow only teams with non-expert users (as suggested by Erik Poll: forbid any participants with a Master's degree); one could force expert users to use a tool that they are not very familiar with; one could have mixed teams with users of different tools; or one could forbid tool developers to participate (except as judges).

Unfortunately, all these suggestions seem to have practical problems (how to get enough participants that are not tool builders or experts; program verification tools often have a steep learning curve; and manuals are not always available). Therefore, we believe that the best workable solution is to consider team composition and tool maturity when judging the solutions. In addition, for future competitions we will explicitly encourage several teams using the same tool to participate, allowing user competitions within the overall tool competition.

To increase the variety of participants, we believe that there should be some reward for taking part in the competition. In particular, if you are not a tool developer, then why would you bother participating in a competition? If there is a winner announced, you can put this on your CV. However, competitions could have so many categories that almost all tools and participants can be judged so that they win a prize. Organising competitions, where participants are invited to contribute to a post-competition publication about the challenges could also provide a motivation.

6 Judging

Judging verification competition solutions is challenging, but it is also very exciting. Solutions are typically judged for their correctness, their completeness and their elegance. While tools may verify if a given implementation is correct with respect to the given specification, determining if a solution is complete and elegant is not so straightforward. Presentation of proofs, degree of automation in verification, annotation overhead, and the extent of verification (e.g., partial vs. total correctness, etc.) are some of the further considerations.

In the first two verification competitions the judges manually inspected the solutions providing subjective results at a presentation during the co-located conference. A follow up conference paper allowed the participants to clean up and revise their solutions for public consumption. In the third, a scoring scheme was applied to each solution and submissions were ranked according to the total sum of points they scored. However, it was noted upfront that "a certain degree of subjectivity in judgement is inevitable and should be considered as part of the game."

Tools, solutions and team member abilities vary greatly, so there are many parameters that play a role when measuring the quality of the solutions. One strategy to aid the judging process is to categorise the tools according to their characteristics and maturity, classifying the results based on these categories. Usability should be measured, qualitatively until better metrics can be found.

6.1 The Role of the Tools in Judging

An important question is whether judging has to involve replaying the solutions in the tools. There are arguments both for and against this. The main goals related to the requirement of tool replay are: punishing fraud, tool unsoundness, and specification inadequacy. While we discount the first issue, in the current state of affairs, the others, especially the third, are of great importance. It is important to keep in mind that there is no canonical requirement formalisation, and tool replay does not bring an ultimate judgement.

If the tool produces an explicit proof object, inspecting it may expose both unsoundness and specification inadequacy. Otherwise, the only way a tool may help is mutation testing. If after changing a part of the requirement (including the program being verified), the tool can still show conformance, there is *a*

probability that the changed part contributes nothing to the problem. This *may* indicate an issue with the tool soundness or requirement adequacy (or both).

The biggest argument against tool replay is the effort, both in installing and running the tools and carrying out the solution analysis as described above. There exists many versions of many verification tools, which can be installed on many different platforms, each using many different plug-ins (all having many versions). Knowing the exact tools that will be used in the competition in advance, especially if a large number of tools participate, is essential. Taking these arguments together, at the current level of competition maturity we would not advocate tool replay, unless doubts in the quality of a solution are present, or if the replay in the tool promises significant benefits in judging the solution (e.g., by advanced proof presentation). Replay could be made easier in the future, if tool developers make their tools available via a web interface, or if virtual machine images could be provided.

6.2 Understanding the Argument

In order to judge the completeness or elegance of a solution it is necessary to understand the argument behind it. Unfortunately, program verification arguments are notoriously difficult to communicate. This applied both to systems that expose an explicit proof object (i.e., a derivation in a certain calculus) and to systems where the user only works with the annotated source code and does not see the logical reasoning behind it.

The explicit proof object is typically too fine-grained, while the annotated source code often does not make the argument structure explicit. Moreover, whenever a tool silently infers a particular fact (a termination measure, for instance), it reduces the burden on the user but may appear as a gap in reasoning to an outsider. In any case, a good portion of knowledge about the background theory implemented in a tool is needed to understand a solution.

A team's approach to solving a problem is often one that the adjudicator themselves would not have used. While this is normal for any problem-solving scenario, we have noted that the solution presented is often a result of the particular strengths and weaknesses of the verification tool used.

While the overhead of adjudicating solutions in many different formalisms is quite high, the benefits are many. While adjudicators will not be an expert in every tool, it is our experience that expert non-users of tools can largely understand the various solutions. Examining solutions for the same challenges in many different verification environments is extremely educational, and experiencing different approaches to solving the challenges (tool-driven or user-driven) is also a fun component of the process.

The enthusiasm of participants, both in terms of the tools that they use and the solutions that they develop is also uplifting. Explicitly scheduling an explanation session where the team members talk an adjudicator through the solution (possible with on-site competitions only) would take full advantage of this enthusiasm and assist the judges in developing a complete understanding of the teams' solutions. Above all, we believe that participants should be encouraged

to clean up their solutions and interact after the competition, to discuss their submissions and to compare the strengths and weaknesses of each tool.

7 List of Recommendations for Organisers

To conclude, we end this paper with a list of recommendations for future verification competition organisers. These recommendations arise from our experience of participating and organising verification competitions, and from our interactions with other competition particpants. We believe that these recommendations will contribute to improved verification competitions in the future.

- Associate the competition with a well-established, regular event.
- Encourage newcomers to participate in the competitions.
- Set up a repository of challenges.
- Remember the goals of competitions, and do not disregard small challenges.
- Ask participants to contribute challenges, and reward them for this.
- On-site verification competitions have their place, do not make all competitions online.
- Encourage discussion between participants about their solutions.
- Record teams during competition participation.
- Judge teams depending on the maturity of their tools and the experience of team members.
- Let team members explain their solutions to the judges.
- Encourage multiple teams using the same tool to participate.
- Invited participants to contribute to a post-competition publication.
- Rotate organisation and participation.

We look forward to further verification competitions and are confident that, as they mature, they will become a major contributor to benchmarking verification tools, improving their capabilities, and extending their usability.

References

1. D. Beyer. Competition on software verification (SV-COMP). In C. Flanagan and B. König, editors, *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and of Analysis Systems (TACAS 2012)*, volume 7214 of *LNCS*, pages 504–524. Springer-Verlag, Heidelberg, 2012.
2. T. Bormer, M. Brockschmidt, D. Distefano, G. Ernst, J.-C. Filliâtre, R. Grigore, M. Huisman, V. Klebanov, C. Marché, R. Monahan, W. Mostowski, N. Polikarpova, C. Scheben, G. Schellhorn, B. Tofan, J. Tschannen, and M. Ulbrich. The COST IC0701 verification competition 2011. In B. Beckert, F. Damiani, and D. Gurov, editors, *International Conference on Formal Verification of Object-Oriented Systems (FoVeOOS 2011)*, LNCS. Springer, 2012. To appear.
3. COST Action IC0701. Verification problem repository. `www.verifythis.org`.
4. J.-C. Filliâtre, A. Paskevich, and A. Stump. The 2nd Verified Software Competition: Experience report. In A. Biere, B. Beckert, V. Klebanov, and G. Sutcliffe, editors, *Proceedings of the 1st International Workshop on Comparative Empirical Evaluation of Reasoning Systems (COMPARE 2012)*, 2012.

5. V. Klebanov, P. Müller, N. Shankar, G. T. Leavens, V. Wüstholz, E. Alkassar, R. Arthan, D. Bronish, R. Chapman, E. Cohen, M. Hillebrand, B. Jacobs, K. R. M. Leino, R. Monahan, F. Piessens, N. Polikarpova, T. Ridge, J. Smans, S. Tobies, T. Tuerk, M. Ulbrich, and B. Weiß. The 1st Verified Software Competition: Experience report. In M. Butler and W. Schulte, editors, *Proceedings, 17th International Symposium on Formal Methods (FM)*, volume 6664 of *LNCS*. Springer, 2011. Materials available at `www.vscomp.org`.
6. C. Lewerentz and T. Lindner. Case study "production cell": A comparative study in formal specification and verification. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software*, volume 1009 of *LNCS*, pages 388–416. Springer, 1995.
7. G. Sutcliffe and C. Suttner. The state of CASC. *AI Communications*, 19(1):35–48, 2006.
8. J. Woodcock. First steps in the Verified Software Grand Challenge. *Computer*, 39(10):57–64, 2006.

Challenges in Comparing Software Analysis Tools for C*

Florian Merz, Carsten Sinz, and Stephan Falke

Institute for Theoretical Computer Science
Karlsruhe Institute of Technology (KIT), Germany
{florian.merz, carsten.sinz, stephan.falke}@kit.edu
http://verialg.iti.kit.edu

Abstract. Comparing different software verification or bug-finding tools for C programs can be a difficult task. Problems arise from different kinds of properties that different tools can check, restrictions on the input programs accepted, lack of a standardized specification language for program properties, or different interpretations of the programming language semantics. In this discussion paper we describe problem areas and discuss possible solutions. The paper also reflects some lessons we have learned from participating with our tool LLBMC in the TACAS 2012 Competition on Software Verification (SV-COMP 2012).

1 Introduction

There is a growing number of tools focusing on analyzing the correctness of C programs using formal methods, e.g., model checkers such as BLAST [4] and SATABS [7], bounded model checking tools such as LLBMC [10], CBMC [6], or F-Soft [9], and symbolic execution tools like KLEE [5].

While all of these tools have similar design goals, comparing them can be cumbersome (Alglave *et al.* [1] seem to confirm this). Tool comparisons in other fields (like SAT and SMT [2]) suggest that annual competitions and the possibility to quickly and easily compare tools act as a major driving force within a research community for developing better tools. This has also been realized by the organizers of the software verification competition SV-COMP 2012 [3], which took place in March/April 2012 for the first time.

2 Challenges

In the following we discuss challenges in comparing automated software analysis tools for C programs and suggest possible solutions for each of these problems.

* This work was supported in part by the "Concept for the Future" of Karlsruhe Institute of Technology within the framework of the German Excellence Initiative.

Challenge 1: *What is a correct program?* While many would say that a *specification* is needed in order to determine whether a program is correct or not, we take a slightly different, more general stance here.

Besides errors with respect to a given specification, a second, equally important class of errors is due to underspecification in the programming language definition, e.g. many aspects of the C programming language are underspecified by the C99 standard [8]. Such underspecification is done intentionally to allow for more freedom in implementing the standard on different computer architectures.

As an example, the result of a signed integer overflow in C is deliberately left unspecified in order to allow not only two's complement, but also alternative implementations like, e.g., a sign plus magnitude encoding.

The C standard carefully distinguishes three different kinds of underspecification: *undefined* behavior, *unspecified* behavior, and *implementation-defined* behavior. Implementation-defined behavior indicates that the semantics is defined by the implementation at hand, which usually depends on the architecture and the compiler. In the C99 standard, unspecified behavior is defined as behavior for which the standard "provides two or more possibilities and imposes no further requirements on which is chosen in any instance".

Last but not least, undefined behavior is the least-specified of the three: broadly speaking, if undefined behavior occurs "anything might happen". Or, as Scott Meyers puts it [11]: undefined behavior might even "reformat your disk, send suggestive letters to your boss, fax source code to your competitors, whatever." Because of this malicious nature of undefined behavior we propose to consider it always as a fault in a software implementation.

Proposal: Any program whose behavior depends on undefined or unspecified aspects of a programming language should be considered incorrect. Important errors in C programs, such as buffer overflows (access to not allocated memory), null-pointer dereferences, or integer overflows fall into this class.

Challenge 2: *Which error classes should be supported?* Annex J of the C99 standard defines an extensive list of diverse cases in which undefined behavior occurs, possibly too extensive to support all of these cases in academic tools. On the other hand, for the sake of comparability, a minimal set of error types that should be supported by all verification tools needs to be defined. Most program analysis tools support checking for the following types of errors:

- Signed integer arithmetic overflow.
- Division by zero.
- Undefined bitwise shift behavior.
- Array out of bounds errors.
- Memory access violations.
- Violation of user provided assertions.

This is by no means a complete list of the errors that different tools can detect (e.g., errors in signed integer shift operations are handled by some tools, but not by others). And even if an error class is handled by a tool, its semantics might be interpreted (slightly) differently.

Proposal: One possibility to make checks of such common error classes available in a larger set of tools is to encode them directly into C source code (e.g., by using a preprocessor), as was done in SV-COMP 2012. See Figure 1 for an example. By using such an encoding, the task of bug finding can be reduced to reachability analysis.

```
1 int foo(int x)
2 {
3   return 1/x;
4 }
```

```
1 int foo(int x)
2 {
3   if (x == 0) {
4 error:
5     // unreachable?
6   } else {
7     return 1/x;
8   }
9 }
```

Fig. 1: Encoding a devision-by-zero-error as a reachability check.

This approach helps tool developers to concentrate on the the tools backend, instead of having to implement all different error kinds. It also ensures that the same errors are interpreted in the same way by all tools being evaluated.

Unfortunately, some properties cannot be expressed in simple C code, e.g., memory access correctness, which requires extensive knowledge about the operating system's state. Furthermore, this approach results in intrusive changes to the benchmark, and we therefore suggest to not encode properties as reachability, but require all tools to explicitly support all error classes.

Challenge 3: *How can we assemble suitable benchmark problems?* Assembling benchmark problems to compare the performance of bug-finding and verification tools is an intricate task. Benchmark problems should be of adequate hardness, be as close as possible to real-world programs, and the kind and location of the error should be unambiguous. Thus, a benchmark program should either contain no bug at all or a single, well-known bug. Benchmarks that contain multiple bugs hinder comparison of the results, as different tools might detect different bugs.

Taking undefined behavior as described in Section 2 seriously, we arrive at the following problem: if there is a program location that can exhibit undefined behavior, the program can do anything after having executed the instruction at that location. It could, e.g., jump straight to any position in the code and cause arbitrarily weird follow-up behavior. Thus, a fully conforming verification tool also needs to be able to detect any kind of undefined behavior, even if only a specific class of errors is part of the benchmark suite.

In practice, undefined behavior can have a wide range of different effects. Null-pointer dereferences will typically result in a program crash, whereas signed integer does so rarely, if ever.

No verification tool known to us actually treats undefined behavior according to the standard ("everything is possible"), but opts for a more specific, practical behavior instead. Common choices for such a more practice-oriented treatment are:

- Treat signed integer overflow according to two's complement, assuming that no side effects occur.
- Assume program termination on division by zero.
- Assume program termination on invalid memory access operations.

But the result of an overflowing signed integer operation could also be modeled as a fresh, unconstrained integer, while still assuming the operation to have no side effects. This essentially reduces undefined behavior to undefined results. Reading from invalid memory locations and division by zero could also be modeled using fresh, unconstrained variables, writing to invalid memory locations could be treated as a no-op. As can be seen, there is a wide range of different options on how to treat undefined behavior, all with their own merits and without a clear winner.

Proposal: Benchmark problems should contain exactly one error or no error at all. Moreover, the intended semantics of all cases of undefined behavior should either be clearly specified, or all benchmark problems should be entirely free of undefined behavior.

Challenge 4: *How to treat interface functions?* We call a function for which no implementation is provided an *interface function*. Typical examples of interface functions are (standard) library functions or operating system calls. The problem here is that the semantics of such a function is not directly available, but has to be provided by additional means. If a benchmark problem uses such an interface function the tool has to handle this gracefully. There are several options how to do this:

1. The function can be treated as exhibiting undefined behavior, essentially resulting in a call to the function being an error.
2. The result of each call to the function can be interpreted as a fresh variable of the appropriate type, assuming that the function does not have any side effects.
3. Similarly, the theory of uninterpreted functions can be used to implement functional congruence, while still assuming that the function does not have any side effects.
4. A clearly specified semantics is built into the tool.
5. Finally, short stubs can be provided that mimic essential aspects of the real semantics.

Proposal: Especially for standard library functions (e.g., `memcpy`, `strlen`, ...) and operating system calls the last two approaches seem to be preferable. If interface functions are built into the tool, their semantics has to be specified precisely. For other functions, if they are guaranteed to have no side effects, the third option (uninterpreted functions) is preferable.

Challenge 5: *How to treat the initial state of a benchmark problem?* Many benchmark problems currently in use are small stand-alone programs extracted from larger programs. While in a stand-alone program the initial state of global variables, for example, is defined to be zero by the C standard, this does not seem to faithfully capture the intended semantics of an extracted benchmark, namely to isolate a fraction of the program relevant for the error at hand, while still allowing all possible behaviors of the full program.

Proposal: If a benchmark problem is extracted from a larger program, false initialization effects should be avoided. This can be achieved, e.g., by mimicking the behavior of the missing program parts by stub initialization functions that initialize those global variables that are not initialized explicitly to non-deterministic values.

3 Software Verification or Bug Finding?

When software analysis tools are compared, the aim of the comparison has to be defined succinctly. The primary question is: Does the comparison focus on software verification[1] or on bug finding? While both give an estimate on safety, there is a notable difference in what kinds of results are to be expected.

Software verification tools should be required to not only return "safe" as an answer, but also provide a proof of safety. Bug finding tools, on the other hand, are usually not able to prove safety, but instead are intended to find as many bugs as possible. For each bug they should provide a valid trace that allows to reproduce the bug when running the program.[2] Note that producing such an error-trace is usually not required for software verification tools.

This difference also becomes visible in the way one typically thinks about soundness and completeness.[3] In formal software verification soundness means: if the tool claims the code to be correct then the program is indeed correct. Completeness here means: if the program is correct then the software verification tool can generate a proof. Software verification tools cannot necessarily provide a program trace for bugs. In bug finding it is the other way round: if a sound bug finding tool reports an error, that error really exists in the program (i.e., there are no false positives). A bug finding tool is complete when: if the code contains a bug, then the tool finds it. Bug finding tools usually cannot provide a (sensible) proof of correctness, even when they can guarantee that there is no bug in the code.

In almost all tool-comparisons there seems to be agreement that unsoundness should be punished harder than incompleteness. If software verification tools

[1] In the following, with software verification we mean formal proofs of a software's correctness.

[2] We consider developing a common, standardized format for error traces an important research goal by itself.

[3] If a tool is sound and complete, this distinction is not important, but in reality most tools are indeed incomplete.

and bug finding tools are mixed up, this cannot be distinguished that easily. Should we take the definitions from the verification community or the bug-finding community? But the difference is not only in the tool, but also how the tool is used. For instance, our bounded model checker LLBMC [10] can be used as a verification tool if bounds are high enough, or as a bug finding tool, if bounds are set low and high code coverage is to be achieved.

Considered as a software verification tool, LLBMC can be used with increasingly higher bounds, until it either proves the property or reaches a time-out. On the other hand, if the bounds are set low, LLBMC acts as a bug-finding tool aimed at finding bugs quickly, but sacrificing completeness (in the sense of missing bugs that could only be found with higher bounds).

We thus believe that any comparison of tools (in particular in competitions) should set a clear focus on either software verification or bug finding, depending on the goals of the competition.

References

1. J. Alglave, A. F. Donaldson, D. Kroening, and M. Tautschnig. Making software verification tools really work. In *Proc. ATVA 2011*, volume 6996 of *LNCS*, pages 28–42, 2011.
2. C. W. Barrett, L. M. de Moura, and A. Stump. SMT-COMP: Satisfiability modulo theories competition. In *Proc. CAV 2005*, volume 3576 of *LNCS*, pages 20–23, 2005.
3. D. Beyer. Competition on software verification (SV-COMP). In *Proc. TACAS 2012*, volume 7214 of *LNCS*, pages 504–524, 2012.
4. D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker BLAST: Applications to software engineering. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(5-6):505–525, 2007.
5. C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. OSDI 2008*, pages 209–224, 2008.
6. E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Proc. TACAS 2004*, volume 2988 of *LNCS*, pages 168–176, 2004.
7. E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Proc. TACAS 2005*, volume 3440 of *LNCS*, pages 570–574, 2005.
8. ISO. The ANSI C standard (C99). Technical Report WG14 N1124, ISO/IEC, 1999.
9. F. Ivancic, Z. Yang, M. K. Ganai, A. Gupta, I. Shlyakhter, and P. Ashar. F-Soft: Software verification platform. In *Proc. CAV 2005*, volume 3576 of *LNCS*, pages 301–306, 2005.
10. F. Merz, S. Falke, and C. Sinz. LLBMC: Bounded model checking of C and C++ programs using a compiler IR. In *Proc. VSTTE 2012*, volume 7152 of *LNCS*, pages 146–161, 2012.
11. S. Meyers. *Effective C++: 55 Specific Ways to Improve your Programs and Designs*. Addison-Wesley, 2005.

Behind the Scene of Solvers Competitions: the *evaluation* Experience

Olivier Roussel
Université Lille-Nord de France, Artois, F-62307 Lens
CRIL, F-62307 Lens
CNRS UMR 8188, F-62307 Lens
roussel@cril.univ-artois.fr

Abstract. In principle, running a competition of solvers is easy: one just needs to collect solvers and benchmarks, run the experiments and publish the results. In practice, there are a number of issues that must be dealt with. This paper attempts to summarize the experience accumulated during the organization of several competitions: pseudo-Boolean, SAT and CSP/MaxCSP competitions.

1 Introduction

Many different kinds of solvers competitions are organized nowadays. The main goal of these competitions is to evaluate solvers in the same experimental conditions. Another goal is to help collecting publicly available benchmarks and also help identifying new solvers on the market. In fact, the actual goal of a competition is to help the community identify good ideas implemented in the solvers as well as strange results which may lead to new ideas.

The most visible result of a competition is a ranking of solvers, which is certainly a good motivation to improve one's solver. However, a ranking is necessarily an over-simplified view of a competition. Indeed, there are several ways to look at the solvers results, and determining which solver is the best one is indeed a multi-objective optimization problem, with an additional complication being that users do not agree on the importance of each criteria. Besides, there are necessarily a number of biases which can influence the rankings such as the selection of benchmarks and the experimental conditions (hardware characteristics, time limits, ...). To sum up, it should not be forgotten that the results of a competition are not an absolute truth, but are just a way to collect data about solvers on a large scale and allow anyone to start his own analysis.

The author's experience with organizing competitions started in 2005 with the pseudo-Boolean (PB) track of the SAT 2005 competition [1], co-organized with Vasco Manquinho. At that time, most of the infrastructure was borrowed from the SAT environment previously developed by Laurent Simon and Daniel Le Berre. The first innovation in this track was the introduction of the *runsolver* tool [2] to improve the timing of solvers and to control more precisely the allocated resources. In 2006, the pseudo-Boolean evaluation became independent, and the competition framework was improved into a system called *evaluation*.

This system was used to organize a large number of competitions since then: pseudo-Boolean competitions (PB05, PB06, PB07, PB09, PB10, PB11, PB12) [1], SAT competitions (SAT07, SAT09, SAT11) and CSP/MaxCSP competitions (CPAI06, CPAI08, CSC09) [3]. *evaluation* is also used by local users to perform their own experiments and was extended to support QBF, Max-SAT, AIG, MUS and WCSP solvers.

Each competition introduced a new challenge and the number of solver types that *evaluation* supports today makes it a very versatile system. This paper attempts to summarize the most important features of *evaluation*, in the hope that this can benefit everyone.

2 System requirements

The first reason for developing *evaluation* was the organization of the pseudo-Boolean competitions. Since the beginning, the pseudo-Boolean competition (PB competition for short) had two tracks: one for the decision problem and another one for the optimization problem. Therefore, beyond the requirements inherent to any competition of a specific kind of solver, the *evaluation* framework was immediately faced to a few more requirements to support different kinds of problems as well as different kinds of solvers. This section lists the main requirements that *evaluation* must fulfill.

Support for partial answers: In the optimization track of the PB competition, the ultimate goal is to find the optimal value of the objective function, and prove that it is actually optimal ("OPT" answer) within the given time limit. However, proving optimality is generally hard and the solver may reach the timeout before it can claim that it has proved optimality. In order to be able to compare solvers which have not been able to give the definitive answer "OPT" within the time limit, it was necessary to be able to interrupt the solvers in a way that allowed them to give the best (partial) answer they had found so far. The implemented solution was to send a SIGTERM signal to the solver to warn it that the time limit was reached, and give the solver a few more seconds to give the best answer it had found that far (in this case, a "SAT" answer indicating that a solution was found, but this solution is not necessarily optimal). Once the grace period is expired, the solver is sent a SIGKILL signal to terminate it. This solution assumes that the solver can intercept the SIGTERM signal, which is easy in most languages. However, this interception is impossible in a few languages such as Java. Fortunately, Java offers a hook to call a function before the termination of the program. Otherwise, solvers must use the time limit parameter provided by the competition environment and make sure that they will terminate gracefully before this limit.

Support for incomplete solvers: The next challenge was to add support for incomplete solvers (local search solvers) in the pseudo-Boolean competition. In the SAT competition (and more generally for a decision problem), support for

incomplete solvers is straightforward because the solver stops as soon as it finds a solution. Therefore, the only difference between a complete and incomplete solvers appears on unsatisfiable instances, where incomplete solvers will reach the time limit without providing an answer[1].

The situation is more complicated in the PB competition (and more generally for an optimization problem) because an incomplete solver will keep searching for a better solution until it reaches the timeout. In order to be able to compare incomplete and complete solvers, it is necessary to get the value of the best solution they found and also, the time used to find this solution. Of course, the competition cannot trust any timing given by the solver. The solution was to require the solver to print a specific line each time it finds a better solution, and let the competition environment timestamp each of these lines. The time required to find the best solution found by the solver (without proving optimality) is called T1 in the *evaluation* framework and allows to compare both complete and incomplete solvers on a common basis.

This approach also allowed to obtain for free a plot of the value of the objective function of the solver current solution, as a function of time. Such a graph gives precious information on the convergence of the solver toward the best solution.

Support for different categories: Basically, a SAT solver is able to solve any kind of CNF. Even if it is specialized for a category of instances such as random ones, it is still able to read instances encoding concrete applications even if in practice, it is unlikely to be able to solve it. In the PB and CSP world for example, the situation is different. Not all PB solvers are able to deal with both decision and optimization problems, with both coefficients which fit in a regular 32 bits integer and coefficients that require arbitrary precision arithmetic, with both linear and non-linear constraints. In the same spirit, CSP solvers are not all able to deal with both extensional constraints, intentional constraints and any kind of global constraint.

Therefore, the environment must support different categories of instances defined upon the characteristics of their constraints. Solvers are registered by their authors in one or more of these categories, which indicates that these solvers are technically able to parse and solve these instances. In the PB competition, there are currently up to 8 categories in each of the 2 tracks.

In the CSP world, dealing with global constraints is a challenge of its own. There are several hundreds of global constraints defined, and many solvers implement only a few of them. It is almost impossible to define in advance categories of instances based on the kind of global constraints they contain: this would require enumerating all subsets of the global constraints appearing in the test set, which would not make sense. The proposed solution here is to define a special "UNSUPPORTED" answer that indicates that a solver has no support for a kind of constraint present in an instance. This allows to identify cluster of

[1] Some incomplete solvers may however answer UNSAT in some cases

solvers which are able to solve the same kinds of constraints and compare them on a common basis.

Verification of answers: One of the most fundamental requirement of a competition framework is to verify the answer given by a solver. Indeed, probably one of the major contribution of a competition is to enhance the global quality of solvers by eliminating incorrect solvers during the competition, which is a strong incentive for writing correct solvers! This verification requires that the solver generates a certificate that is checked by the competition framework.

Ideally, this certificate must be cheap to generate, and cheap to verify. This is generally the case for positive answers of a decision problem. For instance, in most cases, a SAT solver can easily print the model that satisfies the CNF. In contrast, certificates for UNSAT answers are much harder to generate and to check, and is the subject of a specific competition. The only check that is performed for UNSAT answers is to verify that no other solver found a solution. The situation is similar for OPT answers, and the only check performed is to verify that no other solver found a better solution.

The MUS (Minimally Unsatisfiable Subset) competition is a specific case. Certificates are easily generated (they are merely a list of clauses that form a MUS) but harder to verify since it must be checked that the MUS is unsatisfiable and that each proper subset is satisfiable. The *evaluation* framework was extended in 2011 to verify these certificates in a post-processing step.

Data recording: A job is a run of a given solver on one instance. During a competition, it is not uncommon that some jobs do not run correctly, either because of problems with the solver or because of problems in the environment (lack of space on /tmp, interactions with processes left running on the host, ...). It is highly desirable to collect a maximum of information on the jobs in order to be able to analyze what actually happened once a problem is detected. In a sense, we need a kind of flight recorder for solvers.

The *evaluation* system stores a lot of information on the job: the host configuration, the solvers parameters, the instance characteristics and the output of the solver. Besides, *runsolver* regularly saves information on the processes started by the solver. In most cases, this is sufficient to identify problems, in which case jobs just have to be run again. Sometimes, some solvers generate tens of gigabytes of output. To protect itself, the environment must limit the size of the solver output. This is done by *runsolver* which preserves the start and the end of the output. The size limit must be chosen with care because some certificates are huge (e.g. > 16 GB).

Some verifications can be done by the competition organizers, but the ultimate verification can only be done by the solver authors, who are the only ones to know exactly how their solver should behave. This is the reason why all the collected data are made available to the authors before the results of the competition are published, so that they can detect problems that would otherwise remain undetected.

Miscellaneous requirements: A first requirement is that the competition framework should have as little interaction as possible with the solver and especially ensure that nothing slows down the solver. Unfortunately, the only way to ensure this would be to run the solver on a non preemptive operating system, which is incompatible with the normal use of a cluster.

In practice, the solver is run on a traditional Unix system, under the monitoring of *runsolver*. This monitoring process necessarily interferes with the solver (access to main memory and to the processor cache, small consumption of CPU time) but the interference is limited [2] and the benefits of *runsolver* exceed its drawbacks.

Another point is that solvers running on different hosts should not interfere, which may happen when instances are read on a network file system. For this reason, the *evaluation* framework first copies both instances and the solver binaries from the network to a local disk (/tmp) and then starts the solver with every generated data stored on the local disk.

To be sure that the instance is correctly copied to the local disk, and that the correct version of the solver is used, a checksum of each copied file is generated and checked against the fingerprint stored in the database.

At last, the environment allows the solver to report additional information (such as the number of nodes, the number of checks,...) which are recorded by the system.

Parallel solver support: More and more solvers are now designed to use the multicores processors which are available on each machine since several years. The environment must obviously record the wall clock time (WC time) and the CPU time of the solver (see section 4.3) but it must also be able to allocate a subset of the cores to the solver (see section 4.1). When a solver is not allocated the complete set of cores available on a host, the system must ensure that only instances of the same solver will run on that host, in order to at least avoid interferences between two solvers designed by different authors.

3 General architecture

In order to ensure privilege separation, the *evaluation* system is implemented as a client/server system. The server manages the dialog with the database and the log file. It is also in charge of granting a job to each client as well as receiving the job results, checking that the solver answer is consistent with the other answers on the same instance, and recording the results in the database.

The client is in charge of receiving a job from the server, copying the solver and the instance to the local disk (/tmp), construct the solver command line and call *runsolver* to monitor the solver execution. *runsolver* stores its data in files on the local disk. At the end of the solver execution, the client runs a verifier program to check the certificate given by the solver. At last, the client copies the files generated by *runsolver* on the local disk to a central directory shared on the network, and reports the results to the server.

The client is also in charge of allocating the cores to the solver, and ensuring that only instances of the same solver are run in parallel on a node.

The last part of the evaluation system deals with the visualization of the results by the users. This part is currently implemented in PHP. Unfortunately, online generation of pages from the databases is too inefficient (the database is several tens of GB large because it contains the results of many competitions). Therefore, HTML pages are generated in advance and stored in a cache. The drawback of this system is that it limits the number of pages that can be generated, because each of them must be stored on the web server (in compressed form). A new system is planned where the generation of web pages would be done by the browser, which would allow more interaction with the user.

4 Resources Allocation and Limits Enforcement

In this section, we detail the problem of allocating resources to both sequential and parallel solvers in a way that is both efficient and as fair as possible.

4.1 Allocation of Cores

The problem of allocating cores to solvers appears when a cluster of nodes with multicores processors is used, which is always the case with recent hardware. The nodes in the cluster used by *evaluation* have two quad-core processors and 32 GB RAM. Obviously, it is highly desirable to optimize the use of the cluster and run concurrently as many solvers as possible on one host. On the other hand, it is also highly desirable to obtain results that are both reproducible and do not depend on external factors such as the other processes running on the system. Besides, if several solvers are run in parallel, we want to measure times that are almost equivalent to the ones of a solver running alone on the same host. Unfortunately, these objectives are contradictory. As soon as several programs are running in parallel (including the *runsolver* process monitoring the solver), they necessarily compete for access to main memory and more importantly to the various cache levels of the processor.

Some experimentation performed on Minisat, indicated that running 8 sequential solvers in parallel (one core allocated to each solver) induced a 35 % time penalty in average, running 4 sequential solvers concurrently on a node (2 cores allocated per solver) implied a 16% penalty in average and at last running 2 sequential solvers concurrently on a node (4 cores allocated per solver) implied almost no penalty (0.4 %) in comparison of running one single sequential solver per node.

In the SAT 2011 competition, it was decided to run 4 sequential solvers in parallel (2 cores per solver) during phase 1 which is used to select the solvers which can enter the second phase. In phase 2 which is the one actually used for ranking solvers, only 2 sequential solvers were run in parallel on a node (4 cores per solver). Parallel solvers where allocated 4 cores in phase 1 (two solver running in parallel on a node) and 8 cores in phase 2 (only one solver per

node). As explained previously, *evaluation* ensured that only instances of the same solver were run in parallel on a given host.

Clearly, a balance must be found between the precision of the time measurements and the number of solvers run in parallel on the cluster. Given the time constraints and the number of solvers submitted to a competition such as the SAT competition, there is little hope to have enough computing resources to afford running one single solver per node in any case.

4.2 Allocation of Memory

Once the policy for allocating cores to solvers is decided, one must decide of the policy for allocating memory to solvers. The basic policy is to reserve a fraction of RAM to the system and to share equally the rest of memory between the solvers running in parallel. Solvers are not allowed to swap on disk, because this kills the hardware and most importantly gives times which are only representative of the hard disk performances. As an example, solvers in the SAT 2011 competition were allowed to use 31GB divided by the number of concurrent solvers. In phase 1, this amounts to 7.7 GB for sequential solvers and 15.5 GB for parallel solvers. In phase 2, this amounts to 15.5 GB for sequential solvers and 31 GB for parallel solvers. Given policy on core allocation, parallel solvers were allocated twice the memory of a sequential solver!

This looks clearly unfair at first, and actually it is, but on the other hand, parallel solvers necessarily need more memory than sequential solvers. Hence, enforcing the same limit would not be fair either! Clearly, we believe that there is no way to be absolutely fair regarding memory allocation for sequential and parallel solvers. The policy chosen in that competition is not perfect, but can be seen as an indirect way to encourage the development of parallel solvers.

4.3 Allocation of Time

In computer science, there are mainly two distinct notions of time: wall clock time and CPU time. The wall clock time (WC time for short) is the real time that elapses between the start and the end of a computing task. The CPU time is the time during which instructions of the program are executed by a processing unit. On a host with a single processing unit, CPU time and wall clock time are equal as long as the system does not interrupt the program. As soon as a time-sharing system is used on a single processing unit, wall clock time will usually be greater than CPU time, because during some time slices the processor will be allocated to another program. On a host with n processing units, if the program is able to use efficiently each of these units and is not interrupted by the system, the CPU time will be equal to n times the wall clock time. Generally speaking, the CPU time is a good measure of the computing effort, while wall clock time corresponds to the user's perception of the program efficiency.

For sequential solvers, there's a clear agreement that CPU time is the right measure of efficiency since it allows to mostly abstract from the perturbation

caused by the computing environment. For parallel solvers, two different points of view exist.

The first point of view is to consider that only the wall clock time matters, which amounts to considering that CPU resources come for free. This might make sense on a desktop computer where the different cores are idle most of the time. However, this leads to approaches which perform redundant computations, such as some portfolio approaches, and clearly waste computing resources. Besides, the assumption that cores come for free does not make sense in larger environments such as clusters or clouds. Each core must be used efficiently.

The second point of view also consider wall clock time but actually puts the emphasis on CPU time. The motivation is that we expect the parallel solver to distribute the computations equally on the different cores and avoid any redundant computation. Therefore, the CPU time used by a parallel solver should not be significantly greater than the CPU time of a sequential solver and the wall clock time of the parallel solver should tend toward the CPU time of the sequential solver divided by the number of cores. Of course, it is well known that this perfect result cannot be obtained because a parallel solver faces problems that the sequential solver doesn't: synchronization problems, contention on memory access,...

In 2009, there has been strong discussions between the organizers of the SAT competition about which point of view should be taken by the competition. In the end, it was decided to put the emphasis on CPU time in order to encourage the community to use efficiently the available cores and also to be able to compare sequential and parallel solvers on a common basis. This lead to enforcing the same CPU limits for both sequential and parallel solvers. In the end, this was unsatisfactory because it showed only one side of the comparison and completely hid the wall clock time information.

In 2011, it was decided to be more neutral and in fact adopt both point of views. Therefore, two different rankings were set up: a CPU ranking and a WC ranking. The WC ranking is based on wall clock time and was expected to promote solvers which use all available resources to give an answer as quickly as possible. In this ranking, timeout is imposed on the wall clock time. The CPU ranking is based on CPU time and was expected to promote solvers which use resources as efficiently as possible. In this ranking, timeout is imposed on CPU time.

Besides, it was decided to organize only one track mixing both sequential and parallel solvers. Indeed, there's no actual reason to differentiate sequential or parallel solvers. The only thing that matters is their performances, either in CPU time or in WC time. It was expected that parallel solvers would perform better in the WC ranking while sequential solvers would perform better in the CPU ranking. Mixing the two kinds of solvers in a same ranking, either CPU or WC based, allows a mostly fair comparison. If a parallel solver does not perform better than a sequential solver in the WC ranking, there is no point in using it. Conversely, if a sequential solver does not perform better than a parallel solver in the CPU ranking, there is no point in using it.

For a chosen timeout T_o and a number n of available cores, the idea was to have a CPU ranking with a limit on CPU time set to T_o and a WC ranking with a limit on WC time set to T_o. Obviously, it was essential to run one single experiment to get both information. Therefore, sequential solvers were run with a CPU limit of T_o and parallel solvers were run with a CPU limit of $n.T_o$. Since the operating system may suspend the solver execution for some time, we have to select a WC limit which is slightly greater than T_o, otherwise it might be impossible to reach the CPU limit in some cases. Generally speaking, the CPU limit is considered more reliable than the WC limit because it presumably does not depend on the other processes running on the system. A post-processing of the results enforces the same CPU or WC limit to generate the CPU and WC rankings respectively.

Choosing the right WC limit for the ranking is actually extremely difficult. On the one hand, it is clear for sequential solvers that the WC limit should be slightly larger than T_o, let's say $T_o + \epsilon$. The value of ϵ can be chosen relatively large because it is only used to compensate delays that are not caused by the solver. On the other hand, for parallel solvers, it makes more sense to set the WC limit to be equal to T_o. Otherwise, a solver that uses all n cores will hit the CPU limit set to $n.T_o$ after a WC time only slightly greater than T_o, but a solver that leaves some core idle may never hit the CPU limit and only hit the WC limit $T_o + \epsilon$. If ϵ is large, this would imply that inefficient parallel solvers would be granted more WC clock time than efficient parallel solvers. Here, the choice of ϵ compensate delays that are caused in part by the solver itself and therefore should tend to 0.

As an example, in the second phase of the 2011 competition, the experiments were performed with a WC limit set to 5100 s for all solvers. Sequential solvers were allowed to use 5000 s CPU time and parallel solvers on 8 cores had a limit set to 40,000 s CPU time. Results were post-processed to enforce a CPU and WC limit of 5000 s for the CPU ranking and a CPU limit of 40,000 s and a WC limit of 5000 s for the WC ranking.

4.4 Enforcing limits

Once resources are allocated to the solvers, limits on these resources must be enforced. This task is not as obvious as it seems. One important problem is that the most straightforward command for measuring the time of a program (the `time(1)` command) frequently fails for solvers running multiple processes, which occurs as soon as a shell script is used to start the solver. Indeed, this command uses times(2) to display the time statistics of the solver. However, this system call only returns the "resources used by those of its children that *have terminated and have been waited for*". This implies that if, for some reason, the parent process doesn't call wait(2), the resources used by the child will be ignored. This also means that these commands cannot enforce reliable limits for multi-process solvers because the resources used by the child are only reported when it terminates.

runsolver was designed to avoid this trap, as well as some others, and implements several other requirements presented at the beginning of this article. A detailed description of *runsolver* is out of the scope of this paper, but can be found in [2]. We only present here its main characteristics.

runsolver is a Linux specific program and is freely available under the Gnu Public License from `http://www.cril.univ-artois.fr/~roussel/runsolver`. Basically, *runsolver* can be seen as the integration of ulimit(1), time(1) and ps(1) with several improvements. It is called with the command line of the solver to run, and parameters specifying the various limits. Once *runsolver* has launched the solver, it periodically monitors the time and memory consumption of the solver processes by fetching the relevant information from the `/proc` filesystem and summing the resource usages[2]. Sanity checks are performed to identify cases where a parent did not wait for its child. As soon as the solver reaches a resource limit, it is gracefully stopped. Process or thread creation or deletion by the solver are also monitored. Periodically, the list of processes run by the solver is saved in a log file in order to allow a post-mortem analysis of what happened.

Each line printed by the solver can be timestamped to identify how much CPU and wall clock time elapsed since the start of the solver. This is a very convenient feature that allows to learn for example how long the solver took to parse the instance or to learn at what time the solver improved its current solution (for optimization problems).

At last, *runsolver* is able to allocate to a solver a given subset of the host cores (with sched_setaffinity(2)) and is able to deal with solvers that generate a huge amount of output (sometimes several tens of GB) by storing only the start and the end of its output.

Since *runsolver* was designed to avoid requiring any root privilege, it runs as a regular program and slightly compete with the solver for CPU usage and memory access. However, the resources used by *runsolver* are very limited and the perturbation is negligible (see [2]).

5 Rankings

The most visible aspect of a competition is to produce a ranking of solvers, but it should be emphasized that such a ranking can only represent one point of view.

Indeed, there are many different ways to look at solvers, and different users generally have different points of views on the comparison of solvers. One point of view is to consider that the solver able to answer on the greatest number of instances is the best one. This is the point of view adopted in several competitions. It has the advantage to be simple and effective, but of course it is somewhat over-simplified. One may also want to consider the number of solved instances in each family, and prefer solvers which have either a balanced number of solved instances in each family, or inversely prefer solvers which solve the most

[2] Memory of threads of a same process is not added, since they share the same address space.

instances in the family of interest to the user. Some users consider the integration of the solver into a wider system and prefer a fast solver to a solver answering more often but which is slower in average. They may also prefer solvers using in average less memory than competitors with similar results.

Alternatively, one may wish to give an advantage to solvers which use new techniques that are the only ones able to solve some instances. Indeed, the purse scoring [4] integrates this point of view. For each instance, a purse of points is divided between the solvers that gave an answer. When only a few solvers are able to solve an instance, they gain more points. This system has interesting properties, but also some drawbacks (the score of a solver depends on the other solvers for example). Several other scoring methods have been proposed [5,6], each with their own pros and cons.

Several other aspects could or should be taken into account in the rankings. The robustness of a solver, that is, its ability to solve an instance which is close to an instance that it already solves (for example instances obtained by shuffling constraints and variables) is a desirable feature. Determinism, that is, the ability to give the same answer in approximately the same time when the solver is run on the same instance several times, is a feature which is important for the adoption of solvers in industrial applications.

Clearly, there are many criteria to compare solvers and expecting to integrate all these criteria into one single ranking is just an utopia. Therefore, one must accept that a competition ranking is just a way to attract contestants, but that it cannot summarize all the details of the picture taken by the competition.

As a last illustration of this point, let us consider the situation of sequential and parallel solvers. It is clear that sequential solvers must be compared on CPU time. Regarding parallel solvers, WC time is clearly an important parameter. Some users consider that the cores present on their computer come for free and disregard CPU time. In our opinion, this is a mistake. CPU time is a resource of its own, which becomes obvious when the solver is integrated into a larger system. The solution adopted in the SAT 2011 competition is to compare solvers on the two criteria: CPU and WC time, without separating sequential and parallel solvers. This generates a CPU ranking and a WC ranking in which each solver appears. The rationale is that, a parallel solver is of no interest if it does not outperform a sequential solver in WC time, and conversely a sequential solver is of no interest if it does not outperform a parallel solver in CPU time. In practice, it has been actually observed that sequential solvers outperformed some parallel solvers in the WC ranking and that parallel solvers outperformed some sequential solvers in the CPU ranking.

6 Conclusion

This paper presented the principles that governed several competitions (PB, SAT, CSP, MUS,...). It can be seen that several issues must be solved during a competition. Several points are open to discussion. Nevertheless, in the end, organizers must choose their own policy. It should be remembered that a com-

petition does not generate an absolute truth: it merely generates a lot of data that can be analyzed in different ways by the community and that contribute to the improvement of solvers, which is the sole actual goal of a competition.

References

1. Manquinho, V., Roussel, O.: The First Evaluation of Pseudo-Boolean Solvers (PB'05). Journal on Satisfiability, Boolean Modeling and Computation **2** (2006) 103–143
2. Roussel, O.: Controlling a Solver Execution: the runsolver Tool. Journal on Satisfiability, Boolean Modeling and Computation(JSAT) **7** (nov 2011) 139–144
3. Lecoutre, C., Roussel, O., Van Dongen, M.: Promoting robust black-box solvers through competitions. Constraints **15**(3) (jul 2010) 317–326
4. Van Gelder, A., Le Berre, D., Biere, A., Kullmann, O., Simon, L.: Purse-based scoring for comparison of exponential-time programs. Poster (2005)
5. Nikolic, M.: Statistical methodology for comparison of sat solvers. In Strichman, O., Szeider, S., eds.: SAT. Volume 6175 of Lecture Notes in Computer Science., Springer (2010) 209–222
6. Van Gelder, A.: Careful ranking of multiple solvers with timeouts and ties. In: Proc. SAT (LNCS 6695), Springer (2011) 317–328

Author Index