The 1st Verified Software Competition: Extended* Experience Report

Peter Müller, Natarajan Shankar, Gary T. Leavens, Tom Ridge, Thomas Tuerk, Vladimir Klebanov, Mattias Ulbrich, Benjamin Weiß, K. Rustan M. Leino, Rod Chapman, Rosemary Monahan, Nadia Polikarpova, Derek Bronish, Rob Arthan, Eyad Alkassar, Ernie Cohen, Mark Hillebrand, Stephan Tobies, Bart Jacobs, Frank Piessens, and Jan Smans

`www.vscomp.org`

Abstract. We, the organizers and participants, report our experiences from the 1st Verified Software Competition, held in August 2010 in Edinburgh at the VSTTE 2010 conference.

1 Introduction

Research on SAT solving and automatic theorem proving has been boosted by the competitions held in connections with conferences such as SAT, CADE, and CAV. The regular comparisons of tools help the community by exhibiting the practical impact of algorithms and implementation strategies, and help its clients by providing an assessment of the performance of individual tools as well as of the research field overall.

Inspired by this success, participants of the Verified Software Initiative [8] decided to start a program verification competition, which was first organized by Peter Müller and Natarajan Shankar and held at the VSTTE 2010 conference. While the long-term objective is to provide similar benefits to the community like the ATP, SAT, and SMT competitions, the goals for the initial event were much more modest—to create interest among researchers and tool builders in this kind of event, to get an impression of how such an event is received by the community, and to gain experience in designing and carrying out a verification competition.

The competition was explicitly held as a forum where researchers could demonstrate the strengths of their tools rather than be punished for their shortcomings. There were no deliberate attempts to expose weaknesses such as unsoundness or incompleteness of the verification tools, or missing support for certain language features. The organizers presented five small programs and suggestions what to prove about them (such as the absence of run-time errors, functional behavior, or termination). After the presentation followed a four-hour thinking period where no tool use was allowed. After that, the participants had

* The floor brackets mark parts not present in the short version. This text was last updated on January 26, 2011.

two hours to develop their solutions. The participants could work in teams of up to three people, provided that all of them were physically present on site. The physical presence allowed the organizers to interact with the participants and to get immediate feedback about the challenge problems and the organization of the competition.

There was no ranking of solutions or winner announcement. The evaluation committee (Gary Leavens, Peter Müller, and Natarajan Shankar) manually inspected the solutions and pointed out strengths and weaknesses according to the criteria of completeness, elegance, and (reported) automation; these subjective results were presented at the conference to foster discussions among the participants. Not ranking the results allowed in particular a comparison of different verification approaches, whereas a fair ranking would have required standardization and grouping by disciplines (such as automatic vs. interactive or modular vs. whole-program verification).

This setup proved to be successful. Eleven teams participated in the competition and submitted in total 19 (partial) solutions to the five challenge problems (reproduced in Section 2). For this paper, the participants also had the chance to revise or complete their solutions (see Table 1 for an overview). Ten out of 11 original teams report their experiences in Section 3. A number of challenges, common issues, and conclusions are presented in Section 5.

The original problem statements, all team solutions, as well as an extended version of this report are available on the competition web site.

2 The Challenge Problems

In the following, we list the competition problems, slightly edited for brevity. The original problem descriptions included reference implementations in pseudocode and test cases.

Problem 1: SUM&MAX. Given an N-element array of natural numbers, write a program to compute the sum and the maximum of the elements in the array. Prove the postcondition that $\texttt{sum} \leqslant N \cdot \texttt{max}$.

Test Case: With the array 9, 5, 0, 2, 7, 3, 2, 1, 10, 6, N is 10, `max` is 10, and `sum` is 45.

Problem 2: INVERTing an Injection. Invert an injective (and thus surjective) array `A` of N elements in the subrange from 0 to $N-1$. Prove that the output array `B` is injective and that $\texttt{B[A[}i\texttt{]]} = i$ for $0 \leqslant i < N$.

Test: If `A` is 9, 3, 8, 2, 7, 4, 0, 1, 5, 6, then the output `B` should be 6, 7, 3, 1, 5, 8, 9, 4, 2, 0.

Problem 3: Searching a LINKEDLIST. Given a linked-list representation of a list of integers, find the index of the first element that is equal to zero. Show that the program returns a number i equal to the length of the list if there is no such element. Otherwise, the element at index i must be equal to zero, and all the preceding elements must be non-zero.

Team	Tool	Problems solved										Implementation / specification language	Tool web site
		at competition					in the aftermath						
		SUM&MAX	INVERT	LINKEDLIST	N QUEENS	QUEUE	SUM&MAX	INVERT	LINKEDLIST	N QUEENS	QUEUE		
A.Tsyban 1	Isabelle	■	□	■	□	□	■[a]	□	■[a]	□	□	C / Hoare logic	
anonHolHacker 1	HOL4	□	□	□	□	■	□	□	□	□	■[a]	HOL	`hol.sourceforge.net`
Holfoot 1	Holfoot	■	□	□	□	□	■	■	■	■	■	C-like / sep. logic	`holfoot.heap-of-problems.org`
KeY 3	KeY	■	■	□	□	□	■[a]	■[a]	■	■	■	Java / JML(+)	`key-project.org/VSComp2010`
Leino 1	Dafny	■	□	■	□	■	■[a]	■	■[a]	■	■[a]	Dafny	`research.microsoft.com/dafny/`
SparkULike 1	SPARK	■	□	□	□	□	■[a]	□	□	□	□	SPARK	`libre.adacore.com`
MonaPoli 2	Boogie	■	□	◩	□	□	■[a]	■	■	□	■	Boogie	`research.microsoft.com/boogie/`
Resolve 1	Resolve	□	□	□	□	■[b]	■	■	■	■	■[a]	Resolve	`resolve.cse.ohio-state.edu:8080/ResolveVCWeb/`
RobArthan 1	ProofPower	■	□	■	□	□	■[a]	□	■[a]	□	□	HOL	`www.lemma-one.com/ProofPower/index/`
VC Crushers 3	VCC	■	□	■	□	□	■	■	■	■	■	C / VCC annotat.	`vcc.codeplex.com`
VeriFast 1	VeriFast	■	□	■	□	□	■	■	■[a]	■	■	C, Java / sep. logic	`www.cs.kuleuven.be/~bartj/verifast/`

Numeral = number of persons at competition

[a] solution unchanged since competition
[b] solved before the competition

Single entry = language integrating implementation and specification

■ solved □ not solved ◩ substantial partial solution

Table 1. Solutions overview

Problem 4: N QUEENS. Write and verify a program to place N queens on an $N \times N$ chess board so that no queen can capture another one with a legal move. If there is no solution, the algorithm should indicate that.

Thus, with $N = 2$, the result should be empty, whereas with $N = 4$, there should be a legal placement.

Problem 5: Amortized QUEUE. An applicative queue with a good amortized complexity can be implemented using a pair of linked lists, such that the front list joined to the *reverse* of the rear list gives the abstract queue. The queue offers the operations `Enqueue(item: T)` to place an element at the rear of the queue, `Tail()` to return the queue without the first element, and `Front()` to return the first element of the queue. The implementation must maintain the invariant `queue.rear.length` $\leqslant$ `queue.front.length` (prove this). Also, show that a client invoking the above operations observes an abstract queue given by a sequence.

3 The Team Reports

3.1 Team anonymousHolHacker (Tom Ridge)

HOL4 [14] is an interactive theorem prover for higher-order logic, broadly similar to systems such as Isabelle/HOL, HOL Light, and ProofPower. HOL4 has good automated proof support, including powerful equality reasoning (simplification, i.e., rewriting with directed equalities), complete first-order proof search, and decision procedures for decidable fragments of arithmetic. Extensive libraries of theorems covering many common data types and functions are also provided.

Competition Only QUEUE was attempted during the competition. QUEUE was chosen because it was perceived to be the most challenging, although in some ways it is more straightforward than the other questions. INVERT is also relatively difficult, but unfortunately this was not identified by Ridge during the competition, and so no attempt at this problem was made.

The HOL statement uses an abstraction function `abstr` to construct the queue by joining (`++`) the two underlying implementation lists (represented as the pair `impl`):

```
abstr impl = (front impl) ++ (REVERSE (rear impl))
```

All the data types, functions, and required properties given in the problem statement are fairly directly transcribed in HOL. Three very simple arithmetic facts are established, and proof of the required properties then proceeds essentially by case analysis on lists, and simplification, with a few trivial instances of first-order proof (first-order proof with appropriate case splitting and other library lemmas would automatically prove all the required properties outright). Induction is not explicitly needed in the proofs, so that QUEUE is in some ways simpler than the other problems. However, the arithmetic facts and various HOL4 library lemmas about lists essentially are inductive: the simplicity of our proofs (the lack of induction) derives from the maturity of the HOL4 system, especially the automation for arithmetic lemmas, and the extensive libraries of theorems about lists.

The HOL4 solution is given at a relatively abstract level, and no attempt is made to address imperative features such as linked lists and pointer manipulation. The natural approach would be to rephrase the queue functions in a model of an object-oriented language whose semantics was formalized inside HOL4. The proofs would be essentially the same, but there would be significant overhead maintaining various separation-type properties of the two implementation lists. It would be interesting to define the semantics of a simple object-oriented language and investigate this approach.

3.2 Team Holfoot (Thomas Tuerk)

Holfoot is an instantiation of a general separation logic framework inside the HOL4 [14] theorem prover. It is able to reason about the partial correctness of programs written in a simple, low-level imperative language, which is designed to resemble C. This language contains pointers, local and global variables, dynamic memory allocation/deallocation, conditional execution, while loops, and recursive procedures with call-by-value and call-by-reference arguments. Moreover, concurrency is supported by conditional critical regions and a parallel composition operator.

Holfoot follows in the footsteps of the separation logic tool Smallfoot [4]. It uses the same programming language and a similar specification language but gives them a rigorous formal semantics in HOL. As all inferences pass through the HOL4 kernel, the Holfoot proofs are highly trustworthy with respect to the defined formal semantics. Also, while Smallfoot is concerned only with the shape of data structures, Holfoot can reason about their content as well, supporting full functional verification. Holfoot can handle arrays and pointer arithmetic.

Simple specifications, like the Smallfoot examples or a fully functional specification of reversing a singly linked list can be verified automatically in Holfoot. More complicated ones like fully functional specifications of quicksort or insertion into a red-black tree require interactive proofs. These interactive proofs can use all the infrastructure of HOL4.

Competition The Holfoot team consists only of Thomas Tuerk, the developer of Holfoot. Unluckily, only the first example was solved during the competition due to time limitations. This is mainly due to Thomas Tuerk not being familiar with HOL4's arithmetic reasoning infrastructure. INVERT was tried, but not finished during the competition.

Aftermath Since the competition, all problems have been solved using Holfoot. As a separation logic tool, Holfoot is aimed at reasoning about dynamic data structures. Therefore, Holfoot is especially good at reasoning about QUEUE. For other examples, HOL4's infrastructure for defining new predicates and functions was beneficial. INVERT for example uses a newly defined function to translate the original problem into a functional one inside HOL4.

3.3 Team KeY (Vladimir Klebanov, Mattias Ulbrich, Benjamin Weiß)

The KeY system [3] is a verification tool for Java programs. At the core of the system is a deductive prover working in first-order Dynamic Logic for Java (JavaDL). Properties of programs can be specified in JML or OCL, which KeY translates into proof obligations in JavaDL. Specifying directly in JavaDL is also possible.

The KeY system is not strictly a verification condition generator (VCG), but a theorem prover for program logic interleaving symbolic execution of programs, first-order reasoning, arithmetic, and symbolic state simplification, etc. Via its SMT export interface, the system can also use external solvers (such as Z3) to discharge goals.

For programs annotated with requirements and sufficient loop invariants, the system can often find verification proofs automatically. On the other hand, the system does expose an explicit proof object of (relatively) good understandability. The user can provide guidance to the prover by manipulating the proof manually at key points—for instance adding lemmas or instantiating quantifiers.

Competition At the competition, the KeY team consisted of three developers with in-depth knowledge of the system. We used a pre-release of KeY 1.6. During the discussion phase, it quickly became clear that—at the current state of our technology—time constraints alone will not allow solving more than three problems. By the end of the allotted time, we had solved SUM&MAX and INVERT, which fall into the class where KeY is strongest (functional-arithmetical properties).

Both problems could be specified without difficulties in standard JML. The specifications were complete regarding the problem formulation. For SUM&MAX, we have also specified and proven that the program indeed computes the sum and the maximum of the array. KeY found the proof automatically (with one goal discharged by a tweaked strategy setting), and the pure prover running time was about six seconds. Quite some time was wasted on INVERT in search of the loop invariant, which turned out to be simpler than expected. In the proof, it was necessary to invoke Z3 *and* manually instantiate two quantifiers (in the surjectivity precondition). Attempts to solve LINKEDLIST were not successful within the given time limit. We did not attempt N QUEENS or QUEUE.

Aftermath After the competition, complete solutions to the three outstanding problems have been produced, using a development branch of the KeY system [12], which is stronger in handling recursive data structures. An extended variant of JML was used for specification. The solutions to LINKEDLIST and QUEUE are inspired by those of Leino (Section 3.4): dynamic frames in the form of ghost fields are used for framing, and mathematical sequences for specifying functional behavior. The total effort spent was two person-weeks, which included some extensions to the verification system.

A closer look at INVERT The challenging part of the problem was to prove the injectivity of `B`. The goal combines quantifiers with linear arithmetics, which is notoriously difficult for SMT solvers. Their performance in this regard is very sensitive to the syntax of the problem formulation.

The goal is to prove that for any $N > 0$, the injectivity of `B`

$$\forall x, y.\ 0 \leqslant x < y < N \to \mathtt{B[}x\mathtt{]} \neq \mathtt{B[}y\mathtt{]} \tag{1}$$

follows from the inverse relation between the arrays `A` and `B` (which per loop invariant holds after the loop)

$$\forall x.\ \big(0 \leqslant x < N \to \mathtt{B[A[}x\mathtt{]]} = x\big) \tag{2}$$

and the surjectivity of `A` (which is a lemma that the problem description allowed to assume)

$$\forall x.\ \big((0 \leqslant x < N) \to \exists x'.\ (0 \leqslant x' < N) \land x = \mathtt{A[}x'\mathtt{]}\big)\ . \tag{3}$$

KeY currently cannot prove this implication automatically. One can, alternatively, invoke the SMT export feature of KeY and have Z3 discharge the formula. The catch with the latter option is that it only succeeds in the formulation exactly as above. After Skolemizing the quantifiers in (1) (which the automated proof search of KeY typically does), Z3 no longer recognizes the formula as valid.

A quick sketch of the desired proof can be given as follows:

> After Skolemizing (1) and abstracting for clarity from index ranges, we have to show that for any $x_0 \neq y_0$:
>
> $$\mathtt{B[}x_0\mathtt{]} \neq \mathtt{B[}y_0\mathtt{]}\ . \tag{4}$$
>
> Instantiating (3) with x_0, we can rewrite $\mathtt{B[}x_0\mathtt{]}$ to $\mathtt{B[A[}x'_0\mathtt{]]}$ (for some x'_0 with $x_0 = \mathtt{A[}x'_0\mathtt{]}$) and then to just x'_0 with (2). In the same manner, we can rewrite $\mathtt{B[}y_0\mathtt{]}$ to y'_0 (for some y'_0 with $y_0 = \mathtt{A[}y'_0\mathtt{]}$). Thus, we have reduced (4) to showing that $x'_0 \neq y'_0$.
>
> Assuming to the contrary $x'_0 = y'_0$, we can derive $\mathtt{A[}x'_0\mathtt{]} = \mathtt{A[}y'_0\mathtt{]}$ and thus $x_0 = y_0$ (remembering the properties of x'_0 and y'_0), which contradicts our knowledge about x_0 and y_0.

During the competition, we have manually instantiated the surjectivity assumption (3) with x_0 and y_0 respectively. After that, the proof obligation was discharged by Z3. It is also possible to complete the rest of the proof in KeY by also instantiating (2).

3.4 Team Leino (Rustan Leino)

Dafny is an object-based language with built-in specification constructs [9]. To a first approximation, it is like Java (but without subclasses) with Eiffel- or JML-like specifications. Language features that are especially useful when writing specifications include sets and sequences, ghost variables, and user-defined recursive functions. Dafny uses mathematical integers (implemented by big-nums), which avoids overflow errors.

The Dafny verifier statically checks all specifications, language rules (e.g., array index bounds), termination, and other conditions (e.g., well-foundedness of functions). To help it along, a user supplies assertions like method pre- and post-conditions, loop invariants, and termination metrics. The compiler then omits specifications and other ghost constructs from the compiled code. Like VCC, the Dafny verifier is built using Boogie [2, 11], which in turn uses the SMT-solver Z3 [7] as its reasoning engine. The preferable way to develop Dafny programs is in the Microsoft Visual Studio IDE, where the Dafny verifier runs in the background and verification errors are reported as the program is being designed.

Competition Solving SUM&MAX came down to adding a one-line loop invariant.

To solve LINKEDLIST, I associated with every linked-list node a ghost variable whose value is the sequence of list elements from that point onward in the list. To state the appropriate invariant about that ghost variable, one must account for which linked-list nodes contribute to the value, which is done using a common "dynamic frames" specification idiom in Dafny.

The linked list in QUEUE is similar to the one in LINKEDLIST, but stores in every node the length of the remaining list and provides additional operations like `Concat` and `Reverse`. To build an amortized queue from two linked lists, one reversed, is then straightforward using a user-defined function that returns the reverse of a given sequence.

The competition was an adrenalin rush and a race against the clock. I had gone into it hoping to finish all five problems, but ended up with incomplete attempts at INVERT and N QUEENS. In retrospect, I may have finished INVERT had I ignored N QUEENS.

As the author of the tool, I may not be a good judge of its user-friendliness. But for me, I found the immediate feedback from the verifier running in the background useful throughout.

Aftermath The difficulty with INVERT lies in getting the SMT solver to make use of the given surjectivity property. The general trick is state a lemma, an assert statement whose condition supplies the reasoning engine with a stepping stone in the proof. In particular, the lemma will mention terms that trigger reasoning about quantifiers that also mention those terms. In INVERT, the surjectivity property does not contain any terms that can be used in a lemma, so I introduced a dummy function for that purpose.

I found N QUEENS to be the most difficult problem, because it involves verifying the absence of a solution in those cases where the given search strategy does not find one. After some more verbose attempts, I was able to get this down to two lemmas.

3.5 Team SPARKuLike (Rod Chapman)

SPARK is a contractualized subset of the Ada language, specifically designed for the construction of high-assurance software. It has an industrial track record spanning some twenty years, including use in projects such as the EuroFighter Typhoon, the Lockheed-Martin C130J, and the NSA's Tokeneer demonstrator system. The overriding design goal of the language is the provision of a sound verification system, which is based on information-flow analysis, Hoare logic, and theorem proving.

Rather than tackling all the problems in this challenge, I decided to take on the first (Sum&Max), but aiming at a complete implementation and proof to the standard that we would expect for industrial safety-critical code. In particular, the solution offers a complete proof of partial correctness, type safety, and termination. Test cases were also developed that offer a respectable coverage of boundary conditions and structural coverage. The proof of type safety also covers the absence of arithmetic overflow. This was not required by the competition rules, but was felt to be achievable in SPARK through the judicious selection of well-defined ranges for the basic numeric types—a common practice in SPARK. Indeed, failure to specify numeric ranges is normally considered an outright design error in SPARK.

The solution took 107 minutes total, broken down as follows: Planning 5, Design 40, Coding and Proof 50, Compile 1, Test 1, Review and Write-up 10. The very low times for Compile and Test are encouraging—essentially no defects were discovered at this stage. The SPARK Verification Condition Generator produces 18 VCs, of which 14 are proved automatically. The remaining 4 VCs require some additional Lemmas and are completed with the interactive prover.

3.6 Team MonaPoli (Rosemary Monahan, Nadia Polikarpova)

Boogie 2 [11] is an intermediate verification language designed to accommodate the encoding of verification conditions for imperative, object-oriented programs. Boogie [2] is a static verifier that accepts Boogie 2 programs as input and generates verification conditions, which are then submitted to one of the supported theorem provers (the default being the SMT solver Z3 [7]). Several program verifiers, including verifiers for Spec#, Havoc, VCC, Dafny and Chalice, generate their verification conditions by first translating the source program and its specification into the intermediate language Boogie 2 and then transforming that intermediate language program into logical formulae using the Boogie tool. In this competition, we chose to write our solutions directly in Boogie 2, using the Boogie tool and Z3 (version 2.11 during the competition, version 2.15 for the final version) to verify our solutions.

Competition At the competition the MonaPoli team consisted of Nadia Polikarpova and Rosemary Monahan, two people who had just met at VSTTE 2010. Both had used the Boogie tool but primarily as an underlying component of verifiers for other languages. The team worked together and submitted solutions to SUM&MAX and LINKEDLIST.

We attempted LINKEDLIST first. Specifying heap-manipulating programs in Boogie 2 requires explicitly defining the heap, so we defined the linked list by mapping a list cell to its stored value and to the next list cell. Our specification included auxiliary functions which calculated the length of the list, determined if a value was in the list, and returned the value at a particular position in the list. Our main observation from this solution was that while the need to specify the heap is an overhead, it ensures that the specifier has a complete understanding of the program semantics. The solution we submitted at the competition was incomplete as we used two unproved lemmas. Our solution for the paper is complete and proves automatically in about 2 seconds.

Our solution to SUM&MAX was easily specified and automatically verified in less than 2 seconds. Our main observation here was that specifications for small, integer- and array-manipulating programs in Boogie 2 are simple and concise.

We did not prove termination for any of the problems as Boogie 2 does not directly support termination measures. One way around this is to encode termination properties by hand, introducing an auxiliary variable to store the value of the measure at the previous iteration of the loop, a loop invariant that states that the measure is non-negative, and an assertion that the measure has decreased.

Aftermath After the competition, solutions to INVERT and QUEUE were completed.

In INVERT, proving that one array is an inversion of another simply requires the addition of an obvious loop invariant. Proving that an array is injective is more complicated. The main difficulty was making Boogie instantiate the surjectivity precondition.

$$\forall\, \texttt{k}\colon \textbf{int} \bullet \texttt{0} \le \texttt{k} \wedge \texttt{k < N} \Longrightarrow (\exists\, \texttt{i}\colon \textbf{int} \bullet \texttt{0} \le \texttt{i} \wedge \texttt{i < N} \wedge \texttt{A[i] =k})$$

Instead, we introduced a ghost set mirroring all seen values of `A` and loop invariants stating that the set cardinality is exactly `k` (`k` being the loop counter) and that all elements are in $[0; N)$. To this end, we formalized a small theory of sets.

During the solution development we noticed that verification was often helped by introducing auxiliary functions and replacing expressions with function calls (though this does not show with the latest version of Z3).

QUEUE delivered a more interesting experience as theories of sequences and heap allocation were required. These were not difficult to specify but were quite labor-intensive. However, once these theories have been written, it is possible to solve a whole range of similar problems, so the effort is not wasted. Our theory of sequences developed for QUEUE contains several examples of proofs by induction as well as an example of proof by contradiction (lemma `Sequence.zero_count_empty`) which may be of interest to the reader.

When dealing with linked data structures, one typically needs to define inductive properties. We noticed that in order for Z3 to handle them effectively it is important to use induction on structure instead of induction on integers. A case in point is the definition of the value function `at` for the nth cell in the list from LinkedList. The following definition works (showing recursive case only)

```
((item[jj] = at (ii, n)) ∧ (next[jj] ≠ nil)) ⟹
         (item[next[jj]] = at (ii, n + 1)));
```

while the following definition does not (again, recursive case only):

```
(n > 0) ⟹ (at(ii, n) = at(next[ii], n - 1))
```

Verification of the list and queue implementations was also greatly simplified by the fact that both classes are immutable: no advanced techniques for specifying footprints of the methods (such as dynamic frames) were required. Thus, proving that a method call does not change the value of a certain expression could be achieved by asserting that if operands of the expression were allocated before the call they could not be modified by a call to a weakly-pure method. This shows how much easier it is to prove immutable classes (even if method bodies are imperative inside).

Other remarks In many cases during the solution development, Boogie did not respond from the start within a reasonable time, struggling to complete the proof. In such cases the usual debugging technique—inserting **`assert`** statements to determine which facts the tool can infer—is not effective, as there is no indication, which assertion the tool is struggling to prove. Weakening the postcondition was not feasible as the method contained recursion. In this case an alternative technique was successful: we inserted **`assume`** statements for all postconditions at the end of a procedure and then, adding one assertion at a time, worked towards the final proof goal in small steps. If Boogie then took too long to respond, it was struggling with the last added assertion.

With regard to teaching program verification we suggest that using Boogie 2 has an advantage over some high-level languages, as all the advanced object-oriented features (such as heap, class invariants or frame properties) have to be spelled out explicitly in terms of low-level constructs (global variables, pre- and postconditions). When we showed our competition solutions to students who were taking a course on program verification their first reaction was negative. However, when they started verifying their own specifications they quickly realized the benefits of "no magic" behind the scenes. It helped them to understand how the verifier works and hence how to debug their code.

3.7 Team Resolve (Derek Bronish)

Resolve is a tool-supported programming and specification language for full-functional verification of imperative component-based programs [13]. The language emphasizes strict separation of client- and implementer-views of components, providing full modularity both in terms of human comprehensibility and the proof process [?]. The key to this approach is the maintenance of value semantics for all types, so references cannot "leak" across component boundaries [?].

Verification conditions generated automatically from Resolve code may be discharged either by interfaces with third-party provers such as Isabelle and Z3, or by SplitDecision [?], an internally-developed tool that applies theorems of the mathematical theories that pervade the specification language (e.g., strings, finite sets, tuples, etc.).

Competition The Resolve group representative did not originally intend to participate in the competition and has only submitted QUEUE, for which we already had a solution posted to the web.

Most notably, the Resolve solution to this problem (the `StackRealization` of the `QueueTemplate`, viewable online at `http://resolve.cse.ohio-state.edu:8080/ResolveVCWeb`) uses an abstraction to separate the queue from extraneous implementation details such as the nodes and pointers that may comprise the lists' concrete realizations. In other words, the amortized queue is represented as two stacks, which themselves may use a linked-list representation, but the implementation details of the stacks are separated from the proof of the queue implementation. An important tenet of Resolve is that such modularity is required for verification efforts to scale upwards to more complex software systems.

Aftermath Since the competition proper, solutions to all five problems have been composed in Resolve. An important attribute of the solutions, allowing all of the VCs to be discharged either mechanically or simply by hand, is the use of specifier-supplied mathematical definitions to hide quantifiers. For example, the postcondition for INVERT can be expressed as:

```
a.lb = #a.lb and a.ub = #a.ub and
IS_INVERTED_UP_TO(a.ub + 1, #a, a)
```

This states that the bounds of the array are not changed, and the outgoing value of the array is completely inverted with respect to its incoming value. The definition of `IS_INVERTED_UP_TO` is rather complicated and involves a universal quantifier, but this definition never needs to be expanded in order to verify the code. Instead, one simply applies universal algebraic lemmas such as:

$$i \texttt{ = a.lb} \implies \texttt{IS\_INVERTED\_UP\_TO(}i\texttt{, a, b)}$$

How best to design a verification system that allows specifiers to provide such definitions and lemmas, demonstrate the validity of the lemmas as a one-time cost, and then incorporate proven lemmas into its automated reasoning engine is an ongoing research question, which experience in this competition has revealed is important and promising for the future of Resolve.

3.8 Team RobArthan (Rob Arthan)

ProofPower [1] is a tool supporting specification and proof in HOL (Mike Gordon's polymorphic formulation of Church's simple type theory) and other languages, most notably the Z notation, via semantic embeddings in HOL. ProofPower is the basis for an Ada verification system called the Ada Compliance Tool developed for QinetiQ, who use it for verifying safety-critical control software, using Z specifications derived from Simulink diagrams.

Competition For the competition, as I felt that functional programming was rather under-represented at VSTTE, I decided to write recursive definitions in HOL of functional programs and verify those. The resulting "programs" are executable in ProofPower using the rewriting engine, although this is not really a general purpose execution environment.

The conservative extension mechanism used to make the definitions imposes a consistency proof obligation. This proof obligation is discharged automatically for all the examples in the solutions and the syntactic form of the definitions then guarantees termination.

The solutions are modular in the sense that the new functions are defined by combining existing functions, and theorems about those new functions are derived from theorems about their constituent functions. The list searching solution first defines a polymorphic search function with a higher-order parameter giving the search criteria and instantiates it to search for zeroes in a list of integers.

This means that one can do particular calculations in the theorem prover with the results as theorems. I just did this for testing purposes in the competition, but it is an important technique in the application of systems like ProofPower to mathematical and engineering problems requiring highly-assured calculations, e.g., Tom Hales's Flyspeck project uses this kind of technique in HOL Light and Isabelle/HOL.

I was the only ProofPower user at VSTTE at the time of the competition, so I formed a team of one. I am one of the main authors of the system. Given the time available, I chose SUM&MAX and LINKEDLIST as the problems most amenable to the techniques I was using. The other problems could easily be handled in much the same way, but a few more hours would be required.

3.9 Team VC Crushers (Eyad Alkassar, Ernie Cohen, Mark Hillebrand, Stephan Tobies)

VCC is an assertional, first-order deductive verifier for industrial-strength concurrent C (and assembly) code. VCC verification is based on modular two-state invariants, which allow the encoding of a variety of verification disciplines. (There is explicit syntactic support for Spec#-style ownership.) To overcome the restrictions of first-order reasoning, ghost state/code are typically used to maintain inductively defined information (e.g., the reachable nodes of a recursive data structure), with ghost code substituting for prover guidance. (For example, simulation is encoded by maintaining the abstract state as ghost state, with explicit updates to this state witnessing the simulation.) Verification conditions are discharged by an automatic prover (currently, Z3), but there is also a backend connection to Isabelle/HOL. VCC currently verifies only partial correctness (but termination is coming soon).

The VC Crushers team consisted of three persons during competition time, who were joined by a fourth person (Ernie Cohen) afterwards.

Competition SUM&MAX was solved modulo two assumptions related to C's use of bounded (machine) integers. The first assumption was that the sum maintained in the loop did not overflow. This has to either be assumed in the loop, provided as a precondition, or taken into account in the postcondition. The second assumption was of a nonlinear arithmetic property that Z3 could not handle effectively for bounded integers. In addition to the required postcondition, we also proved that the result for the maximum is a bound for the individual elements and that the function result is the summation of the array elements. LINKEDLIST was also fully solved during the contest, but using an overly complex list specification with many superfluous invariants in the list data structure. INVERT was attempted during the competition, and was partially but not completely finished.

Aftermath The remaining problems were solved after the competition.

For SUM&MAX, we discovered that the nonlinear arithmetic assumption could be proven by Z3 for unbounded integers (which helps explain why other Z3-based verifiers did not run into the same problem). The work-around in our solution is to "guide" Z3 by asserting the unbounded property (essentially making it available as a lemma). We also removed the no-overflow assumption by weakening the postcondition to say that either the result is correct or the (unbounded) sum overflows.

In INVERT, we use a ghost map parameter *inverse* to the function to encode surjectivity of the input array `A`. The central hint to the prover to show the postconditions on the output array `B` is to rewrite `B[`j`]` to `B[A[`*inverse*`[`j`]]]`; getting Z3 to do this automatically required using a custom trigger. Alternatively, we could have explicitly provided a hint (by mentioning a term of the form `B[A[`*inverse*`[`j`]]]`) where needed.

In our contest solution for LINKEDLIST, we used an overly complex list implementation (one that maintains the reachability relation through arbitrary first-order surgery on lists). However, this complexity is not needed for the contest problems, so we re-did the verification using a much simpler list implementation (used also for QUEUE).

The main difficulty in N QUEENS is how to express the non-existence of the solution when the search procedure returns false. Our C implementation uses arrays (and destructive updates) to work on the board. VCC does not allow assertions to quantify over heaps (for reasons related to logical consistency), so we instead used maps (a mathematical abstraction) to reason about the solution space (with the same encoding as for boards). To express that there is no solution in a certain search state, we state that all solutions sharing the same prefix as the current board are inconsistent (i.e., have a queen i capturing a queen j).

In VCC, reading an object requires evidence that it still exists. In most cases (including typical sequential code), this is done by owning the object. When the object has to be shared, this is usually done by owning a ghost object (called a claim) whose invariant guarantees the existence of the object in question. Manipulating these claims increases the annotation burden, but allows the data to eventually be destroyed. On the other hand, this problem tacitly assumes garbage collection, since the code creates shared data with no way to reclaim it. We verified a version of the problem that does its own memory management (essentially consuming data passed into functions); the solution verifies quite conveniently using ownership, but does not allow reuse. As expected, the solution had to make additional assumptions (or preconditions) to make sure that memory allocations do not fail and that the queues do not grow too large.

3.10 Team VeriFast (Bart Jacobs, Frank Piessens, Jan Smans)

VeriFast is a verifier for single- and multithreaded C and Java programs. It takes as input C or Java source files, annotated with pre- and postconditions, loop invariants, definitions of inductive data types, fixpoint functions, recursive separation logic predicates, lemma functions, as well as some proof steps in specially marked comments. It outputs either "0 errors found" or both the source location of a potential error, and a symbolic execution trace leading up to the error, with the symbolic heap, the symbolic store, and the path condition at each execution step. These can be browsed conveniently in the VeriFast IDE. The tool is intended to be sound: modulo bugs in the tool, an output of "0 errors found" implies memory safety, data-race-freedom, and compliance with user-provided assertions.

Our experiences with our verification-condition-generation-based verifiers Spec-Leuven and VeriCool had left us frustrated with the unpredictable, and often very bad, performance of the SMT solver on the quantifier-rich queries generated by those tools, mainly to deal with heap effect framing. When designing VeriFast, we put a very strong premium on predictable performance. To deal with heap effect framing, we copycat Smallfoot [4] and perform symbolic execution with memory represented as a separating conjunction of "heap chunks", i.e., separation logic predicate applications. The SMT solver is used only to reason about the arguments of the heap chunks, i.e., the data values. Furthermore, we avoid general quantification in specifications—in fact, it is currently not supported. The only quantifiers that are made available to the SMT solver are those that axiomatize the inductive data types and fixpoint functions (primitive recursive functions over inductive data types); these behave very predictably. The approach pays off: VeriFast's typical sub-second verification times enable a comfortable interactive annotation-insertion experience.

Competition One member of our team, Bart Jacobs, participated at VSTTE and the competition. The first problem he tackled was SUM&MAX. He first tried a Java version, since we have some automation for dealing with arrays in Java. Unfortunately, however, our automation proved quite incomplete. Bart had so much trouble dealing with the complex terms involving `take`, `drop`, `append`, etc. that described the inductive list representing the contents of the array, that he decided the automation was working against him, so he switched to C where VeriFast has no special support for arrays. A C array can be described using a simple recursive predicate. This allowed him to complete SUM&MAX, but by then the competition was more than halfway through. Along the way, however, he also struggled with an incompleteness in the theory of multiplication and inequalities in the version of Z3 that he was using.

He then moved to LINKEDLIST, which, since based on a nice recursive data structure, was a piece of cake for VeriFast.

Finally, he started on QUEUE, which, it seemed, should have been easy for the same reason. However, again, VeriFast's automation started acting up. Sharing of immutable data structures can be expressed in VeriFast using fractional permissions [5]. VeriFast automatically splits and merges fractional chunks as necessary—usually. In this case, it did not, so some time-consuming contortions were necessary to get the sharable linked list implementation finished, not leaving time to complete other problems.

The main conclusion that we took away from the competition is that automation is evil :-). Nonetheless, we will of course continue to work on more and better automation.

Aftermath We have now completed all problems. QUEUE was fairly easy, once the right encoding of sharability was found. (Quantify over the list's fraction, or over each field's fraction separately? Quantifying over each field's fraction works better.) Completing INVERT and N QUEENS required developing quite a bit of theory, which was labor-intensive but possible in VeriFast. For example, for INVERT we proved surjectivity of `A` from injectivity and boundedness.

4 Solution Verbosity

A proposal to measure textual verbosity as a benchmark criterion in verification was recently made in [10]. Inspired by this proposal, a number of teams have measured the verbosity of their solutions in three categories:

1. Code. This category measures the program source code that is compiled and executed. Does not include "ghost" code. Does not include test harnesses or main methods (as the latter were not required in the competition).
2. Requirement Annotations. Requirement annotations constitute the specification of the program. They assure the behavior of the program module towards its environment. They are visible externally and cannot be changed easily. They are the reason for performing verification. Typical requirement annotations are pre- and postconditions of public (non-helper) methods.

Framing conditions of such methods—even though not *explicitly* mentioned in the problem descriptions—have requirement character. In QUEUE a data structure invariant was also a requirement.

3. Auxiliary Annotations. Auxiliary annotations exist solely to guide the proof search. As long as they satisfy their purpose, auxiliary annotations can be changed anytime without notice. Lemmas, intermediate assertions, loop invariants, and ownership clauses are typically members of this category.

It should be noted that the distinction in requirement and auxiliary is in many parts relative to a module boundary. Since the problems treated in this competition were very small, the implied module boundary should be clear from the context.

Verbosity metrics were collected with a Perl script initially released in connection with [10]. The script tokenizes the input, taking into account the lexical conventions of C-derived programming and specification languages. The tokens are assigned to one of the above categories according to the mark-up inserted into the files by solution authors. The results of the measurement are given in Table 2. The script and the marked-up solutions are available on the competition web site.

5 Conclusions

Results of the competition SUM&MAX was the easiest problem, solved by everybody attempting it. INVERT—while not very difficult—challenged the systems' quantifier handling in presence of linear arithmetic. LINKEDLIST provided differentiation in reasoning about heap data structures. N QUEENS and QUEUE were perceived by most as outside the achievable in the competition time frame. Altogether, N QUEENS was probably the most difficult problem, combining complex reasoning and a difficulty to express when there is no legal solution.

The issue of theory reasoning A common issue in the competition was the battle to solve the arising SMT problems. In the majority of cases, the solvers were successful. When they were not (this was most notable in INVERT), the stress for the users was high. In the aftermath, we have seen a wide range of more or less elaborate workarounds for such cases. Better ways for the user to guide the proof search (and for the system to give feedback) are needed. The inference speed, on the other hand, was generally deemed adequate in this competition.

The issues of ADTs and modularity For LINKEDLIST and QUEUE, participants have produced solutions of different flavors of modularity. An interesting solution class were behavioral specifications, i.e., the ones completely separating interface and implementation. In LINKEDLIST, such separation required introducing additional methods for constructing lists, even though they did not contribute to the computation required in the problem. A desirable property of specifications is a clear syntactic separation of interface and implementation (at best, keeping them in separate files), as it makes understanding modularity concepts easier.

Team	Solution verbosity (tokens) code / requirement annotations / aux annotations														
	Sum&Max			Invert			LinkedList			N Queens			Queue		
anonHolHacker	–	–	–	–	–	–	–	–	–	–	–	–	231	172	976
KeY	70	120	110	50	195	52+	90	151	233	228	253	799+	429	571	319
Leino	80	42	11	52	234	99	122	162	194	285	176	418	472	417	210
MonaPoli	84	12	12	58	125	458	82	315	41	–	–	–	779	1909	1868
Resolve	138	221	71	109	228	57	126	499	48	309	711	90	292	138	0
RobArthan	48	173	285	–	–	–	121	68	548	–	–	–	–	–	–
VC Crushers	80	148	208	44	241	54	73	129	114	193	341	148	504	997	154
VeriFast	80	66	450	47	273	1834	59	94	359	269	644	3110	430	463	422

"+" indicates additional non-textual user interaction.

Table 2. Solution verbosity metrics

Concerning the use of abstract data types (ADTs), there is still a gap between different reasoning traditions. Foundational systems like HOL have elaborate and well-established ADT theories, while verification systems for imperative and OO code mainly use ADTs in an ad hoc manner. A systematic connection between the two realms remains a challenge.

Judging solutions and competition organization The competition made apparent that even a qualitative evaluation of solutions, with an informal setup and no ranking, is not an easy task. Solutions varied greatly in their requirement formalization and proof methods. Understanding the details of a solution (let alone validating it with a tool) requires a significant effort from an evaluation committee. Helpful in this regard could be holding a dialogue with the developers, or using a structured questionnaire such as [6]. Certain merits of a solution can be effectively measured [10] (the web version of this report contains statistics on solution verbosity), while others (e.g., elegance) remain subjective. Discussing verification solutions is not as standardized or automated as judging other reasoning tool competitions, but it is extremely instructive.

Other suggestions concerning organization were to include more advanced programming concepts (e.g., concurrency), to allow remote participation thus opening the competition to a wider public, or to assign a separate time slot to each individual problem to achieve a clearer differentiation.

Relevance of the competition The competition (and its aftermath) has shown that all systems are—in the hands of an experienced user—capable of solving any problem. At the same time, already the very “simple” problems posed have exposed many practical issues with current verification tools. These issues are typically not thematized by the way we judge progress in program verification today, i.e., by how big a project can be verified with essentially unlimited resources. The competition with its limited time slot offers a very useful complementary perspective on verification’s way to wide practical use.

An afterword from the organizers The first Verified Software Competition exceeded the expectations of its organizers. We were impressed by the interest the competition received and by the enthusiasm of the participants, which is also demonstrated by the effort spent in the aftermath of the competition to solve the remaining problems. There was a strong encouragement to continue organizing such events. We hope the competition becomes a recurring part of the VSTTE conference and contributes to the Verified Software Initiative.

References

1. B. Adcock. *Working Towards The Verified Software Process.* PhD thesis, Department of Computer Science and Engineering, The Ohio State University, 2010.
2. R. Arthan and R. Jones. Z in HOL in ProofPower. *BCS FACS FACTS*, 2005-1.
3. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO 2005*, volume 4111 of *LNCS*, pages 364–387. Springer, 2006.

4. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2007.
5. J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO 2005*, volume 4111 of *LNCS*, pages 115–137. Springer, 2006.
6. R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *POPL*, 2005.
7. COST Action IC0701. Verification problem repository. `www.verifythis.org`.
8. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS 2008*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
9. C. Hoare, J. Misra, G. T. Leavens, and N. Shankar. The verified software initiative: A manifesto. *ACM Comput. Surv.*, 41:22:1–22:8, October 2009.
10. K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR-16*, volume 6355 of *LNCS*, pages 348–370. Springer, Apr. 2010.
11. K. R. M. Leino and M. Moskal. VACID-0: Verification of ample correctness of invariants of data-structures, edition 0. In *Proceedings of Tools and Experiments Workshop at VSTTE*, 2010.
12. K. R. M. Leino and P. Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In *TACAS 2010*, volume 6015 of *LNCS*, pages 312–327. Springer, 2010.
13. P. H. Schmitt, M. Ulbrich, and B. Weiß. Dynamic frames in Java dynamic logic. In *FoVeOOS 2010*, volume 6528 of *LNCS*. Springer, 2010.
14. M. Sitaraman, B. Adcock, J. Avigad, D. Bronish, P. Bucci, D. Frazier, H. Friedman, H. Harton, W. Heym, J. Kirschenbaum, J. Krone, H. Smith, and B. Weide. Building a push-button RESOLVE verifier: Progress and challenges. *Formal Aspects of Computing*, pages 1–20, 2010.
15. M. Sitaraman, S. Atkinson, G. Kulczycki, B. W. Weide, T. J. Long, P. Bucci, W. D. Heym, S. M. Pike, and J. E. Hollingsworth. Reasoning about software-component behavior. In *ICSR-6: Proceedings of the 6th International Conerence on Software Reuse*, pages 266–283, London, UK, 2000. Springer-Verlag.
16. K. Slind and M. Norrish. A brief overview of HOL4. In *TPHOLs 2008*, volume 5170 of *LNCS*, pages 28–32. Springer, 2008.
17. B. W. Weide and W. D. Heym. Specification and verification with references. In *Proceedings OOPSLA Workshop on Specification and Verification of Component-Based Systems (SAVCBS)*, October 2001.