The COST IC0701 Verification Competition 2011

Thorsten Bormer[6], Marc Brockschmidt[1], Dino Distefano[2,3]. Gidon Ernst[4], Jean-Christophe Filliâtre[7,8], Radu Grigore[2], Marieke Huisman[5], Vladimir Klebanov[6], Claude Marché[7,8], Rosemary Monahan[9], Wojciech Mostowski[5], Nadia Polikarpova[10], Christoph Scheben[6], Gerhard Schellhorn[4], Bogdan Tofan[4], Julian Tschannen[10], and Mattias Ulbrich[6]

[1] RWTH Aachen, Germany
[2] Queen Mary, University of London, UK
[3] Monoidics Ltd., UK
[4] Universität Augsburg, Germany
[5] University of Twente, The Netherlands
[6] Karlsruhe Institute of Technology, Germany
[7] LRI, France
[8] INRIA Saclay, France
[9] National University of Ireland Maynooth, Ireland
[10] ETH Zürich, Switzerland
http://foveoos2011.cost-ic0701.org

Abstract. This paper reports on the experiences with the program verification competition held during the FoVeOOS conference in October 2011. There were 6 teams participating in this competition. We discuss the three different challenges that were posed and the solutions developed by the teams. We conclude with a discussion about the value of such competitions and lessons learned from them.

1 Introduction

A program verification competition was organized as part of the Formal Verification of Object-Oriented Software (FoVeOOS) conference held in Torino, Italy in October 2011. The conference was initiated by the COST Action IC0701, whose topic is advancing formal verification of object-oriented software. One of the tasks pursued by the Action is to set common goals and to develop common benchmarks for program verification tools. The competition aimed—in contrast to larger comparative case studies—to evaluate the usability of verification tools in a relatively controlled experiment that could be easily repeated by others.

The competition was organized by Marieke Huisman, University of Twente, Netherlands, Vladimir Klebanov, Karlsruhe Institute of Technology, Germany, and Rosemary Monahan, National University of Ireland Maynooth, Ireland. All three organizers have an extensive background in program specification and verification, and they have actively contributed to the development of different verification tools.

B. Beckert, F. Damiani, and D. Gurov (Eds.): FoVeOOS 2011, LNCS 7421, pp. 3–21, 2012.

The competition was inspired by, and had a format similar to, the VSComp competition [12] held at VSTTE 2010: up to 3 people could form a team, all participants had to be physically present, and teams could use any verification system of their choice. The event took place the afternoon before the conference officially started. 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 w.r.t. the implementation. In contrast to the VSComp event, a fixed time slot was assigned for each of the three challenges provided. This setup was chosen in order to increase precision of the tool comparisons.

The three different challenges were the following: (1) MaxElim: finding the maximum in an array by elimination, (2) TreeMax: finding the maximum in a tree, and (3) TwoEq: finding two duplets in an array. In addition, a fourth challenge (Cycle) was presented for teams to address outside the competition. This challenge was to determine if a given linked list contains a cycle. The challenges were chosen with the idea that they should be tough, but doable within the given time frames. Thorsten Bormer and Mattias Ulbrich, both Karlsruhe Institute of Technology, Germany, helped select and test the challenges.

For this report, participants were also given the possibility to improve their solutions. A record of all submitted solutions as well as an extended version of this report is available at the competition web site (see front page).

The remainder of this report is structured as follows. First the different teams and tools that participated in the competition are briefly introduced in Section 2. Then, Sections 3, 4, and 5 present the different challenges and the solutions provided by the different teams. Finally, Section 6 presents an overview of solutions submitted both during and after the competition, gathers most interesting observations and conclusions that were made by the organizers and participants, and makes some suggestions about running verification competitions in the future.

2 Participating Teams and Tools

The six participating teams used six different verification tools: the KeY system[1] [2], Dafny[2] [13], KIV[3] [15], jStar[4] [6], Why3 [3] (with the Krakatoa frontend [7]), and AProVe [9]. This section briefly describes these tools, and the background of the team members.

Team KeY. The KeY system [2] is a highly automated, explicit-proof-object theorem prover for Java programs based on Dynamic Logic. Recently, the KeY system started its second life – the current development version is based on explicit heap representations and dynamic frames [11,16]. In this version of KeY, program specifications are written using JML* – a KeY-specific modification of

[1] http://www.key-project.org
[2] http://research.microsoft.com/projects/dafny/
[3] http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/kiv/
[4] http://www.jstarverifier.org

JML to accommodate the idea of dynamic frames. This was the main version of the KeY system used in the competition, however, an older version of KeY based on static frames was also successfully used in solving the first challenge.

The KeY team consisted of two members: Wojciech Mostowski (postdoc at University of Twente) and Christoph Scheben (PhD student at Karlsruhe Institute of Technology). Mostowski is an active developer and user of the KeY system for the last 10 years. Scheben is a recently started PhD student developing the theory and extending the KeY system to reason about information flow properties in Java programs.

Team Dafny. Dafny [13] is a programming language with built-in specification constructs. A Dafny program consists of classes, which contain variables and methods. Methods can be equipped with annotations in the form of pre- and postconditions, inline assertions, loop invariants and termination measures for loops and recursion. Specifications may contain user-defined recursive functions, as well as ghost variables and ghost code. Language features such as sets, sequences, and inductive data types are useful both in specifications and in executable code.

The Dafny verifier statically checks all user-supplied annotations, as well as memory safety properties (such as the absence of null dereferences or array accesses out of bounds), well-foundedness of recursive functions and termination of methods. The verifier is built on top of the Boogie [1] platform and works by generating verification conditions, which are discharged by a proof engine of choice, usually the SMT solver Z3 [5].

At the competition, the Dafny team consisted of two PhD students from ETH Zürich: Julian Tschannen and Nadia Polikarpova. Both team members are novice users of Dafny, however with extensive experience in other Boogie-based verification tools.

Team KIV. KIV [15] is a tool for formal system development and interactive verification. It is based on many-sorted higher-order logic and structured algebraic specifications. KIV supports reasoning about programs written in two languages: 1) abstract programs that contain while loops, nondeterminism and recursive procedures, which operate on arbitrary algebraic data types, and 2) Java programs. The calculus is based on sequents and symbolic execution/wp-calculus for programs. KIV also implements a temporal logic.

KIV has a user-friendly interface with specification graphs and explicit proof trees. Proof automation is achieved by a set of heuristics (e.g., for quantifier instantiation) and efficient compiled rewriting.

Team KIV consisted of two PhD students – Gidon Ernst and Bogdan Tofan, who have both worked with the KIV tool for about two years – and Gerhard Schellhorn, who is one of the main developers of the tool, with many years of experience.

The solutions of the KIV team with full proofs are available online [5].

[5] http://www.informatik.uni-augsburg.de/swt/projects/cost-competition-2011/

Team jStar. jStar is a verification tool based on separation logic that aims at verifying object-oriented programs written in Java. It verifies that programs meet specifications which are provided by the user in the form of method pre- and postconditions. Loop invariants are computed automatically by means of abstract interpretation. The jStar tool is geared towards reasoning about the heap and defers all other reasoning (such as arithmetic reasoning) to the SMT solver Z3.

The jStar tool is built on top of coreStar, a generic language-independent back-end intended for building verification tools based on separation logic. The two essential components that jStar brings together are: a theorem prover for separation logic which embeds an abstraction module for defining abstract interpretations; and a symbolic execution module for separation logic. Both of these components are tailored to object-oriented verification.

Reasoning about arrays was built into jStar only the day before the competition (and thus had not been thoroughly evaluated). As a result, the jStar team did not develop complete solutions to the challenges considering arrays – but used the experience from the competition to find out how they had to improve their support for arrays.

At the competition, the jStar team consisted of Dino Distefano and Radu Grigore from Queen Mary, University of London. Distefano has been working on the development of jStar since 2008, while Grigore is a postdoctoral research assistant working on the project since 2010.

Team Why/Krakatoa. The Why platform [7] is an environment for deductive program verification. It provides a rich specification language for modeling program behavior, and a verification condition generator. The verification conditions are passed to various backends (involving formula transformers/simplifiers, type encoders and pretty-printers) allowing a large set of automated or interactive external provers to be called. It also provides several front-ends to deal with input programs written in mainstream languages such as C (via the Jessie plugin of Frama-C [14]) and Java (with the Krakatoa front-end[6]).

During the competition, the team used Why3 [3], the last major version of Why, and the Krakatoa front-end. The team had only one member, Claude Marché, INRIA Saclay and LRI, France. Marché is an active developer of the C and Java front-ends of Why and Why3 since 2004. Improved solutions were written after the competition together with J.-C. Filliâtre, and are available as part of the ProVal Web gallery of certified programs[7].

Team AProVe. AProVE [9] is a fully automated termination and complexity analysis system with front-ends for several programming languages such as Haskell, Prolog and Java. It builds upon the power of techniques developed for termination analysis of term rewriting systems (TRS) over the past 30 years by using a non-termination-preserving translation from the input problem to a TRS.

[6] http://krakatoa.lri.fr

[7] http://proval.lri.fr/gallery/cost11comp.en.html

For the competition, the team used AProVE's Java frontend [4], which can also be accessed online[8]. The system currently only analyzes full programs, so each challenge solution needed to be accompanied by a routine to generate (random) inputs corresponding to the given pre-conditions. Please note that AProVE *only proved termination properties* for the presented examples.

The termination prover AProVE was used by Marc Brockschmidt, a PhD student working primarily on static analysis of Java programs in AProVE.

3 Challenge 1: Finding the Maximum in an Array

```
public static int max(int[] a) {
    int x = 0;
    int y = a.length-1;

    while (x != y) {
        if (a[x] <= a[y]) x++;
            else y--;
    }
    return x;
}
```

Fig. 1. Search by elimination

Time: 60 minutes
Given: A non-empty integer array `a`.
Challenge: Verify that the index returned by the method `max()` given in Fig. 1 points to an element maximal in the array.
Motivation: This challenge is an instance of Kaldewaij's Search by Elimination [10], where an element with a given property is located by eliminating elements that do not have that property. The challenge was selected as it involves a relatively simple but interesting invariant, expressing that the maximal element is in the remaining search space rather than maintaining the maximal element found so far.
Results: Teams using tools that supported array data structures found the solution to this problem straightforward. Teams KeY, Dafny, KIV and Why/Krakatoa successfully specified and verified the pre- and postcondition of the `max` method. They successfully stated a loop invariant for the main `while` loop. The first part of the invariant is simple: it relates the search space bounds `x` and `y` and the bounds of the input array `a`. The second part of the loop invariant, concerned with maximality, was more difficult to express and verify. Fig. 2 gives an overview of this invariant part, showing quite some variations. Apart from the invariant, all teams have found the right termination measure $\mathtt{y} - \mathtt{x}$.

Team KeY. One particular thing the KeY team found difficult to get right in the invariant was the disjunction of the two cases under each quantifier. The first intuition was that both conditions always hold. However, since it is difficult to establish in the loop invariant which of the two indices is changed by the loop, a disjunction, rather than a conjunction, is correct and at the same time sufficient to prove the final property. The KeY system proves this program fully correct (including integer overflow checks)[9] in around 10 seconds.

[8] http://aprove.informatik.rwth-aachen.de/eval/JBC-Nonterm/

[9] The overflow checking option of the KeY system was only used in this task to show that this is possible. Overflow checks are skipped in the rest.

KeY
```
(\forall int i; i>=0 && i<=x; a[i]<=a[x] || a[i]<=a[y]) &&
(\forall int j; j>y && j<a.length; a[j]<=a[x] || a[j]<=a[y]);
```

Dafny original
```
invariant ∀ i • 0 ≤ i ≤ x ⟹ a[i] ≤ a[x] ∨ a[i] ≤ a[y];
invariant ∀ i • y ≤ i ≤ a.Length - 1 ⟹
  a[i] ≤ a[x] ∨ a[i] ≤ a[y];
```

Dafny revised
```
invariant ∀ i • 0 ≤ i < a.Length ∧
  a[i] > a[x] ∧ a[i] > a[y] ⟹ x < i < y;
```

KIV
$$\exists k.\ x \leq k\ \wedge\ k < y\ \wedge\ k < \#ar \wedge ar[k] = max(ar)$$

Why
```
\forall integer i;
   0 <= i < x || y < i < a.length ==> a[i] <= max(a[x],a[y]);
```

Fig. 2. Invariants for MaxElim (relevant parts)

Team Dafny. The first challenge did not present any problems for the Dafny team. The initial solution was achieved in 25% of the allotted time. It contained five loop invariants, which were essentially a more verbose version of the revised solution that was developed after the competition. It also specified a termination measure through a `decreases` clause, which turned out to be redundant, as Dafny can infer simple termination measures automatically.

Team KIV. The solution of the KIV team models the underlying array data structure as an algebraic array, by actualizing the parameter type of arrays in the KIV library with natural numbers. Array indices are natural numbers too, rather than integers.

After the competition, the team also solved this challenge using KIV's Java calculus [17]. The program is encoded in compilable Java and the proof additionally shows that during execution of the program, no ArrayIndexOutOfBoundsException occurs. The proof structure is identical to the abstract proof, but formulas look slightly more complex, since additional information regarding type and heap access safety is necessary.

Team Why/Krakatoa. The first challenge did not provide any difficulties for the Why/Krakatoa team. Around 15 minutes were enough to write the Why3 version and to prove it correct. Another 15 minutes were sufficient to transform it into a solution for the annotated Java code.

Team AProVe. To analyze the given problem, the AProVE team added a routine to create a random integer array. Then, AProVE could directly prove the resulting program to terminate in under 5 seconds. To achieve this, the `max` routine is automatically translated into a TRS with built-in integers [8] consisting of two *rules* (here simplified for presentation):

$$\mathsf{f}(\mathsf{Array}(l), x, y) \to \mathsf{f}(\mathsf{Array}(l), x+1, y) \mid x \geq 0 \wedge y \geq 0 \wedge y < l \wedge x < l \wedge x \neq y$$
$$\mathsf{f}(\mathsf{Array}(l), x, y) \to \mathsf{f}(\mathsf{Array}(l), x, y-1) \mid x \geq 0 \wedge y \geq 0 \wedge y < l \wedge x < l \wedge x \neq y$$

In term rewriting, a rule $\ell \to r$ can be applied to a term t if there is a substitution σ such that $\ell\sigma = t'$ for some subterm t' of t. Then the application of the rewrite rule results in a variant of t where the subterm t' is replaced by $r\sigma$.

For the example, the two generated rules closely match the two possible loop traversals. The first argument of f represents the input array, for which only the length l is encoded here. Termination is easily, and automatically, proven using a polynomial interpretation corresponding to the measure $2l - x + y$, which decreases in each rule.

Interestingly enough, the loop invariant $x < y$ is not needed to show termination, as x and y are used as array indices and are thus implicitly bounded by 0 and the length of the array.

4 Challenge 2: Finding the Maximum in a Tree

Time: 90 minutes
Given: A non-empty binary tree, where every node carries an integer.
Challenge: Implement and verify a program that computes the maximum of the values in the tree.
Motivation: The challenge was constructed by the organizers to explore how tools handle heap data structures that are not lists. The challenge, nonetheless, did admit a reasonably simple specification with an abstract sequence, map, or similar data type, as it did not involve properties such as the ordering of elements in a tree. Another aspect not tested was data structure mutation.
Results: Within the time slot allocated during the competition, only the KIV team provided a full solution to the problem (and the AProVe team showed termination). However, after the competition, all teams worked out a solution to TreeMax.

Team KeY. In KeY, the solution to TreeMax is based on dynamic frames [11,16]. Due to the linked structure of the tree and the recursive implementation of the `max` method, the KeY team chose to specify both the heap structure of the tree, and a flat representation of it. The heap structure definition states that trees cannot be cyclic, while the flat representation of the tree as a finite sequence of integers disallows infinite trees.

Figure 3 shows the relevant part of the resulting JML* specification. The heap structure is specified with the ghost field `fp`, which denotes the set of locations making up the footprint of the tree, and an invariant that structures this footprint correspondingly. The `accessible` clause provides a measure proving well-foundedness of the recursively defined invariant.

The integer payload of the tree is packed into a sequence in a natural way: the head of the sequence is the node of the tree, the left sub-tree follows, and then the right. The length of the sequence is the measure proving that the recursive call to the method `max` terminates.

```
public class Tree {
  private int value; private /*@ nullable @*/ Tree left, right;

  /*@ ghost \locset fp; invariant fp ==
      \set_union(this.*, \set_union(
        left==null? \empty : \set_union(left.*,left.fp),
        right==null? \empty : \set_union(right.*,right.fp)));

      invariant left != null ==>
        (\disjoint (this.*, left.fp) && left.\inv);
      ...
      ghost \seq seq; invariant seq ==
        \seq_concat(\seq_singleton(value), \seq_concat(
           left==null? \seq_empty : left.seq,
           right==null? \seq_empty : right.seq));
      accessible \inv : fp \measured_by seq.length; @*/
```

Fig. 3. The KeY team's solution for TreeMax (excerpt)

What turned out to be the most challenging part of this task for the KeY team was the actual proof. The boundedness of the invariant and the structure of the footprint is proved automatically and very quickly. The first part of the top-level specification, a universal quantifier stating that the result is greater than or equal to all the elements in the tree is also quite straightforward. Up to this point, the KeY system finds all proofs automatically and within one minute. However, the system has difficulty with the second main property: the existential quantifier that states the presence of the result in the tree. In the automated proof search mode the proof starts to grow uncontrollably regardless of the prover settings used. In the end, substantial manual interaction was required to guide the prover through the important cases (`left` and `right` sub-trees being `null` or not) and to give it the right instantiations for the existential quantifier. This manual interaction with the prover (over 200 interactions in total) was where most of the time was spent on this challenge. The rest of the proof was done automatically and took less than a minute to finish.

Team Dafny. Memory footprints of linked data structures are commonly described in Dafny using dynamic frames [11]. According to this idiom, the `Tree` class is extended with a ghost field `Repr`, which stores the set of all nodes in the subtree with root `this`. The ghost field serves multiple purposes: on the one hand, it is used to describe footprints of functions and methods, thus taking care of the frame problem; on the other hand, it serves as a termination measure for any recursion on subtrees. For the latter use, the data structure is required to be acyclic. Dafny does not support class invariants, where the acyclicity property could be stated. Class invariants are simulated by defining a predicate `valid`, and using it in pre- and postconditions of class methods.

To facilitate functional specifications, another ghost field `Values` is defined to denote the set of all values stored in the subtree. Using this field, it is easy to specify that method `max` returns a value from the tree that is larger than all other values. A straightforward recursive implementation for `max` was provided; the only auxiliary annotation needed to verify this implementation was the termination measure.

In the rush of the competition the Dafny team forgot to specify that the return value must be present in the tree. The team added the omitted piece of specification in the final version, and Dafny was able to verify it without any further modifications. The final version also contains a constructor: a method that establishes the `valid` predicate without requiring it. Adding such a method to any class is important in order to ensure consistency of the class invariant.

Team KIV. The difficult task in the second challenge is to specify a proper tree structure within a heap H. The KIV team used a lightweight embedding of separation logic into HOL. This is part of the KIV library. The embedding encodes heap assertions as heap predicates of type heap $\rightarrow$ bool. It contains a straightforward specification of binary trees in the heap, using a recursive function tr : ref $\times$ tree $\rightarrow$ (heap $\rightarrow$ bool). The heap predicate tr(r, t) states that pointer r is the root of a binary heap tree that corresponds to an algebraic tree t. Algebraic trees are a free data type with two constructors: leaf and branch(t_0, a, t_1). Predicate tr is specified as

$$\begin{aligned}&\mathrm{tr(r,\ leaf)(H)} \leftrightarrow \mathrm{emp(H)} \wedge (\mathrm{r} = \mathrm{null})\\ &\mathrm{tr(r,\ branch(t_0,\ a,\ t_1))(H)} \leftrightarrow\\ &\qquad \exists\ \mathrm{r_0,\ r_1}.\ (\mathrm{r} \mapsto \mathrm{node(r_0,\ a,\ r_1)} * \mathrm{tr(r_0,\ t_0)} * \mathrm{tr(r_1,\ t_1)})\mathrm{(H)}\end{aligned}$$

Predicate emp is true for the empty heap only, and maplet $r \mapsto \mathrm{node}(r_0, a, r_1)$ defines a singleton heap consisting of exactly one node at address r, which holds an element a (actualized with integers in the following) and pointers r_0 and r_1 to the left and right subtree respectively. Crucially, separation logic's star operator $*$ enforces that each tree node resides in a different part of the heap, which ensures the tree shape of the heap structure.

With these preliminaries, total correctness of a recursive procedure MAX(r; i, H) was proven. This procedure returns the maximum value of the tree stored under reference r, in output variable i (the semicolon separates input parameters from reference/output parameters). The proof obligation is:

$$\mathrm{r} \neq \mathrm{null} \wedge (\mathrm{tr(r,t)} * \mathrm{p})\mathrm{(H)} \rightarrow \langle\!|\mathrm{MAX(r;\ i,H)}|\!\rangle\ ((\mathrm{tr(r,t)} * \mathrm{p})\mathrm{(H)} \wedge \mathrm{i} = \mathrm{max(t)})$$

In the formula, $\langle\!|\alpha|\!\rangle\varphi$ is KIV notation for the weakest precondition wp(α, φ) of program α for postcondition φ. Therefore, the goal asserts that all executions of MAX terminate without changing the input tree and that i stores $max(t)$ at the end. The maximum function $max(t)$ was defined by structural recursion over the algebraic tree. Predicate p is a universally quantified predicate variable allowing an abstraction from everything else on the heap beyond the tree structure (p

can, e.g., be instantiated with the empty heap emp). Thus, the generalized goal is directly provable by induction over the size of t. When applying the induction hypothesis for the right (left) subtree, p is instantiated with the original p plus the root cell plus the left (right) subtree. KIV's heuristic applies one instance of the induction hypothesis automatically, the second instance has to be given manually. The proof is simple and has six interactive steps out of 39 steps in total.

Team jStar. The implementation from the jStar team consists of one class whose fields are final, private, and initialized by the constructor. The method `max` is coded as follows:

```
int max() { int r = value;
            if (left != null && left.max() > r) r = left.max();
            if (right != null && right.max() > r) r = right.max();
            return r; }
```

The main objective of the specification is to keep track of the maximum of each tree. The predicate $\mathit{Tree}(t, \{\max = m\})$ denotes that the Java reference t points to a tree whose maximum value is m.

$$\begin{aligned} &\mathit{Tree}(t, \{\max = m\}) \iff \\ &\quad (t = \mathsf{nil} \wedge m = 0) \vee (t \neq \mathsf{nil} \wedge \mathit{NonEmptyTree}(t, \{\max = m\})) \end{aligned} \tag{1}$$

$$\begin{aligned} &\mathit{NonEmptyTree}(t, \{\max = m\}) \iff \exists\, l\ lm\ r\ rm\ v, \\ &\quad t \overset{\texttt{left}}{\longmapsto} l * t \overset{\texttt{right}}{\longmapsto} r * t \overset{\texttt{value}}{\longmapsto} v * \\ &\quad \mathit{Tree}(l, \{\max = lm\}) * \mathit{Tree}(r, \{\max = rm\}) * \\ &\quad m = \max(v, \max(lm, rm)) * v \geq 0 \end{aligned} \tag{2}$$

In general, one can associate abstract records with objects. In this case, the record has exactly one field, max. Verification is easier if one assumes a lower bound on possible values. In this case, the bound 0 was chosen, but also the smallest representable integer could have been chosen. Background axioms specify the function max.

Given this setup, jStar verifies the following specifications.

$$\begin{aligned} &\{\, v \geq 0 * \mathit{Tree}(l, \{\max = lm'\}) * \mathit{Tree}(r, \{\max = rm'\}) \,\} \\ &\langle \mathit{init} \rangle (v, l, r) \\ &\{\, \mathit{NonEmptyTree}(\mathsf{this}, \{\max = \max(v, \max(lm', rm'))\}) \,\} \end{aligned} \tag{3}$$

$$\begin{aligned} &\{\, \mathit{NonEmptyTree}(\mathsf{this}, \{\max = m'\}) \,\} \\ &\texttt{max}() \\ &\{\, \mathit{NonEmptyTree}(\mathsf{this}, \{\max = m'\}) * \mathsf{return} = m' \,\} \end{aligned} \tag{4}$$

Specification (3) ensures that the constructor preserves absence of sharing and value non-negativity when building the tree. More importantly, it keeps track of

the maximum value for each tree that is constructed in the program. Specification (4) ensures that the tree remains allocated and that the returned value is indeed the maximum reachable.

During the competition the jStar team was only able to verify (3). After the competition, the team were also able to verify (4). This could not be proved during the competition as, at that time, jStar could not send extra axioms to Z3, such as the ones for max, which are necessary to prove (4).

Team Why/Krakatoa. For the Why/Krakatoa team, the tree data structure made the second challenge significantly difficult. As a first step, proving a Why3 version of the problem was very useful, since such a data structure can be defined using an algebraic datatype, as follows:

```
type tree = Null | Tree int tree tree
```

Predicates for testing membership of a value `v` in a tree and for checking if a value is greater than or equal to all elements of a tree, can be defined recursively:

```
predicate mem (v:int) (t:tree) = match t with
  | Null -> false
  | Tree x l r -> x=v \/ mem v l \/ mem v r

predicate ge_tree (v:int) (t:tree) = match t with
  | Null -> true
  | Tree x l r -> v >= x /\ ge_tree v l /\ ge_tree v r
```

Given this model of trees, one can provide annotated code for the problem. The team decided to code an auxiliary subprogram, having an accumulator as an extra argument, to take care of empty trees:

```
let rec max_aux (t: tree) (acc: int) =
  { true }
  match t with
  | Null -> acc
  | Tree v l r -> max_aux
      l (max_aux r (MinMax.max v acc))
  end
  { ge_tree result t /\ result >= acc }
```

```
let max (t: tree) =
  { t <> Null }
  match t with
  | Null -> absurd
  | Tree v l r ->
     max_aux l (max_aux r v)
  end
  { ge_tree result t }
```

The postconditions declare that the result is greater than or equal to all the elements of the given tree. This is incomplete, since it is also necessary to express that the result is itself an element of the tree. This is indeed easy to specify using the predicate `mem`. However, during the competition the team made a mistake, and used conjunctions instead of disjunctions in the definition of `mem`, thus failing to prove correctness of the fully specified program. The solution presented here is the one that the team developed during the competition. The ProVal gallery provides a different solution that is both simpler (without auxiliary functions) and complete (with a postcondition that the result appears in the tree).

To prove correctness of `max_aux`'s postcondition, the team used the following lemma:

```
lemma trans: forall t:tree, x y:int.x >= y /\ ge_tree y t -> ge_tree x t
```

Verification conditions are all proved, using a combination of provers, including the interactive prover Coq for the above lemma. The Coq proof script is a few lines long and proceeds by induction over the tree structure.

The ProVal gallery also provides solutions in Java and C. In these cases, complex predicates must be defined in order to specify that the given tree is well-formed, i.e., is a finite tree properly terminated with null pointers as leaves. However, there is no need to specify that there is no sharing in the subtrees: a recursive traversal for finding the maximum is also correct in the case of sharing.

Team AProVe. Again, the AProVE team first added a routine to randomly create a tree. Then, AProVE could automatically translate the `max` method into a TRS. Next, integer comparisons that are not relevant for termination are automatically filtered out, leaving only rules of the form $\mathsf{f}(\mathsf{Tree}(l, r)) \to \mathsf{f}(l)$ and $\mathsf{f}(\mathsf{Tree}(l, r)) \to \mathsf{f}(r)$. These can easily be proven to terminate, as the size of the considered tree decreases strictly in each step. The fully automatic termination proof takes about 15 seconds, where a majority of the time is spent on proving termination of the routine generating a random tree.

5 Challenge 3: Finding Two Duplets in an Array

Time: 90 minutes
Given: An integer array `a` of length $\mathtt{n} + 2$ with $\mathtt{n} \geq 2$. It is known that at least two values stored in the array appear twice (i.e., there are at least two duplets).
Challenge: Implement and verify a program that finds two such values. You may assume that the array contains values between 0 and $\mathtt{n} - 1$.
Motivation: This challenge is a popular "job interview-style" question, but we are not aware of its origin. The challenge was selected as it requires complicated array reasoning, specifications, and invariants.
Results: Most teams solved this problem within or shortly after the deadline for the competition. The KeY team provided a complete solution after the competition finished but before the end of the conference.

Team KeY. The key point in the solution of this challenge is to be able to specify the number of occurrences of a value in the array (and reason about it). In JML, this can be done with the help of the `\sum` operator, a special quantifier that provides an arithmetic sum over the quantified elements.

First, the team defined a method to count the number of occurrences of a given value in the array with the following simple specification:

```
//@ ensures \result ==
//@   (\sum int i; 0<=i && i<a.length; a[i] == value ? 1 : 0);
static int /*@ pure @*/ countAcc(int[] a, int value) {...}
```

The implementation of the method uses a simple loop annotated with appropriate JML specifications to count the occurrences. KeY proves this method correct automatically within a few seconds.

The team then used this method to both implement *and* specify the top-level method that finds the two values that are each duplicated in the array. The solution was developed for a slightly more general case, where it was not assumed that the values in the array are all between 0 and `n`−1. Instead, only the existence of two pairs of duplicates from an arbitrary range of values `smin..smax` (`smin+1<smax`) is required by the precondition of the top-level method. This not only makes the specifications more elegant but also makes the correctness proof easier for the tool to complete.

The implementation of the top-level method is different from that of other teams in that it iterates over all potential array values (given a priori by the `smin..smax` range) and not over the array elements. For each potential value, it calls `countAcc` to get the count of the value's occurrences, and terminates once a second duplicated value has been found. The most difficult part is expressing the invariant for the iteration loop. It makes a case distinction over how many duplicate values have been found so far and specifies a corresponding condition about the elements yet-to-be found. The proof for the top-level method requires some minor interactions (more or less obvious quantifier instantiations), while the rest is done automatically within 10 seconds.

Team Dafny. The main difficulty of the third challenge was to express the precondition that the array has at least two duplicate pairs, and to get the verifier to make use of this fact so that it concludes that the program always succeeds. The team went for a functional approach to specification and formulated the precondition as follows: the array has a duplicate pair, and if that pair is removed from the array, it still has a duplicate pair. To this end, recursive functions `has_duplicates` and `first_duplicate` are defined on sequences.

For the reason of time constraints, the team decided to write the implementation in terms of these functions as well. Dafny makes this possible through a construct called *function method*: a recursive function that must be free of specification-only constructs and thus can be used in executable code.

Note that the implementation uses sequences instead of arrays, because the former are easier for Dafny to reason about. In the opinion of the team, this is not a limitation, as the implementation is still executable.

The postcondition of the program is expressed more abstractly: through the number of occurrences of both results in the initial sequence. To make the program verify, three inductive lemmas were needed. Those lemmas connect the number of occurrences of an element in a sequence to the notions membership, duplicates and removal from a sequence, respectively.

Team KIV. The KIV team first understood (incorrectly) that the task was to compute two *indices* m, n with a duplicate element in the array ($ar[m] = ar[n]$). Therefore, the total correctness of an algorithm FINDDUP$(ar; m, n)$ that finds such a pair of indices was proved initially.

The algorithm uses two nested loops to find the right positions. The outer loop runs through the array using an index m. The inner loop sets *done* to

true if it finds an index $n > m$ with a duplicate. The invariant of the outer loop asserts that no duplicate exists below m, and that the *done* flag of the inner loop indicates $ar[m] = ar[n]$. To verify the inner loop, the precondition is generalized (weakened) from $n = m + 1$ to "no duplicate for $ar[m]$ below n". Then well-founded induction over $\#ar - n$ and symbolic execution of one inner loop body is sufficient to finish the proof.

After verifying this algorithm, the team realized that the intended task was rather to verify an algorithm that computes two duplicate *values* of the input array. Fortunately, both the first implementation and its correctness proof could be reused to come up with the right solution.

A second procedure FINDDUPSND$(ar, k; m, n)$ was then defined, which gets an additional input value k and computes the indices m, n of a duplicate value different from $ar[k]$. The program and the proof for this second algorithm are almost identical to the first. The final theorem that solves the challenge then just combines the two results to prove that executing both procedures sequentially finds the two required duplicates.

Team Why/Krakatoa. The solution that the Why/Krakatoa team implemented during the competition first defines an auxiliary function that given an array a and an *optional* value o, returns the two indexes of a duplet in a whose value differs from o (if any). The main program first calls the auxiliary program without the optional argument o, and then calls it a second time with the argument o initialized to the value found during the first call. The solution is correct whatever the values stored in the array (they do not have to be between 0 and $\mathtt{n} - 1$).

The code is too large to be presented here; it is found in the ProVal gallery. The code for the auxiliary program consists of two nested loops and is not intended to be computationally optimal. Suitable loop invariants can be found without major difficulty. The proofs are obtained using automatic provers. During the competition, the fully proved Why3 program was obtained in around 60 minutes. Java and C versions, as well as an alternative solution in Why3, were implemented after the competition.

Team AProVe. The AProVE team solved the problem by enumerating all pairs of values of the array using two nested loops (using counters `i` and `j` bounded by the array length) and then searching for duplets. AProVE could translate the method to rules of the form $\mathsf{f}(\mathsf{Array}(l), i, j) \to \mathsf{f}(\mathsf{Array}(l), i + 1, i + 2) \mid l \leq j \wedge l > i + 1$ and $\mathsf{f}(\mathsf{Array}(l), i, j) \to \mathsf{f}(\mathsf{Array}(l), i, j + 1) \mid l > j \wedge l > i$. The second rule corresponds to one iteration in the inner loop, in which j is incremented while it is smaller than the array size. The first rule corresponds to the case that the counter j of the inner loop has reached its bound. The rule then encodes that i is incremented, j is reset to $i + 1$ and then a first iteration of the inner loop is performed. Termination of the generated TRS can be proven easily and automatically.

6 Wrap-Up, Conclusions and Future Competitions

Solution Overview. This report discusses the three challenges and the solutions developed during the verification competition organized as part of the FoVeOOS conference in October 2011. Figure 4 summarizes the outcome of the competition.

The "revised" column of Figure 4 also records solutions to the 4th, "take-home" challenge (Cycle), which is mentioned in the introduction. After the competition, the teams Dafny, KIV and Why/Krakatoa submitted solutions for this challenge[10].

	Solutions			Time, % of slot			Revised			
Team (# members)	MaxElim	TreeMax	TwoEq	MaxElim	TreeMax	TwoEq	MaxElim	TreeMax	TwoEq	Cycle
KeY (2)	■	◪	◪	67	100	100	■	■	■	□
Dafny (2)	■	◪[a]	◪[b]	25	83	110	■	■	■	◪
KIV (3)	■	■	◪[b]	75	44	122	■	■	■	■
jStar (2)	□	◪	□	100	100	100	□	■	□	□
Why3 (1)	■	◪	■	25	100	67	■	■	■	◪
AProVE (1)	T	T	T	–	–	–	T	T	T	□

[a] incomplete specification, fixable w/o extra proof hints
[b] full solution shortly after deadline
T verified termination only

■ solved □ not solved ◪ substantial partial solution

Fig. 4. Solution overview

One of the main developers of Dafny, Rustan Leino, has also attempted the challenges, out of competition. He reported that the first three caused him no particular problems (spending approximately 20, 40 and 90 minutes on them, respectively). Studying the solution of the Why/Krakatoa team to TreeMax inspired him to simplify his own. Leino also solved Cycle in about 8 hours. His solutions are now available as part of the Dafny test suite. We encourage others in the verification community to try the challenges and report back to the competition organizers.

The solutions to Cycle submitted by team KIV, team Why, and by Leino verify the efficient "tortoise and hare" algorithm (attributed by Knuth to Floyd). Team KIV also verified Brent's algorithm, which is another efficient cycle detection method.

[10] Available in the extended version of this paper.

Participants' Observations. All teams reported that participating in the competition had been a positive experience and an incentive to continue development of the tools. Taking part in the competition allowed them to get a good overview of the power and usability of verification systems other than the ones with which they were most familiar. Also, because of the requirement to be physically present, the competition provided a good opportunity to interact with developers of competing tools.

During the competition several teams realized the importance of obtaining good feedback from the tool, both upon syntax and specification errors, as well as on failed proof attempts. Developing the solutions under a certain time pressure made these requirements much more obvious than usual.

For the jStar team, the competition gave a clear indication on parts of the tool that had to be improved: jStar's libraries are very basic, and the support for arrays has to be developed further, as the current implementation seems to have some unwanted interactions with Z3.

For the AProVE team, the challenges addressed during the competition were atypical (AProVE usually addresses more complex termination problems). However, the experience of the competition inspired the team to explore how AProVE's fully automated nature might be used to find termination measures that are usable for other tools.

Organizers' Observations. The organizers observed the importance of gathering a bank of challenges well in advance of the competition so that they could be evaluated with respect to their difficulty and their suitability for a wide selection of verification tools. This evaluation was particularly important when determining the length of time to allocate to each challenge.

With respect to the solutions submitted to each of the challenges, the organizers were surprised at the variation of loop invariants used in MaxElim and were interested to see that most teams completed the challenge in less than three quarters of the allotted time. TreeMax brought greater difficulty to teams, with user interaction required by many of the tools and only one team achieving a solution within the allotted time. This re-affirmed the suspicion that verification of programs involving linked data structures is still not a straightforward task for many verification tools. In TwoEq, it was no surprise that most teams divided the challenge into subproblems and took advantage of their tool's support for modular verification to compose the overall solution.

When evaluating the different solutions, the organizers observed that expert non-users of tools can understand the solutions. However, they also observed the importance and the difficulty of communicating program verification proofs. Most tools try to do something in this respect: Why3 supports the user in producing a LaTeX report with key lemmas and some statistics; KIV can process proofs for browsing on the web, however the proofs could be more informative; KeY offers a nicely annotated proof tree, but in the tool only; and Dafny has a very clean annotation syntax[11]. All these aspects help to make the understanding of

[11] An assessment of solution verbosity is available in the extended version of this report.

the proof easier. However, it would be a worthwhile exercise for tool developers to come together to combine and extend the different approaches.

The competition also made it clear that certain tool features help to specify programs and construct proofs. For example, both KIV and Why3 can define and reason easily about arbitrary ADTs, while Dafny has very good built-in support for sets and sequences. However, it is evident that techniques such as systematic refinement between abstract and implemented data types, and invariant generation are not yet adopted in mainstream program verification tools.

Design of Future Verification Competitions. The competition also provided some further ideas about the format of future verification competitions. Unsurprisingly, the outcome of this kind of verification competition depends heavily on the ability and experience of the human proof engineer(s). While this cannot be avoided completely, it is helpful (a) trying to balance the experience and skills between teams, (b) encouraging participation of several teams with the same tool, and (c) trying to attract non-developer teams. On the other hand, it needs to be stressed that tool performance must be measured in terms of *usability* and not just raw deduction power alone. A suggestion that we consider worth investigating is to record the participants as they interact with their tools and later collect their comments on the solution-finding process.

The teams and the organizers appreciated that dedicated time slots were given to challenges. Teams felt that this forced them to work on the problems together, which was beneficial, because sharing ideas reduces the probability of getting stuck on a wrong path. Moreover, it also gives the possibility to pursue two alternative approaches to the same problem in parallel, when one is not sure which one will work.

One of the risks of a verification competition is that the choice of challenges favors a particular tool or approach (in fact, the Dafny team remarked that the challenges did not address modifying the state of complex data structures, which are more difficult to handle in Dafny). An alternative format that would address this issue, would be to ask that each participating team contribute a challenge that they can handle well, and that they believe might be a challenge for the other participants. This would ensure that each team could submit at least one solution to a challenge, and these solutions would provide a good benchmark with which to compare other team solutions.

We conclude that program verification tools are mature enough now to have verification competitions. However, because of the more open nature of program verification problems, and the importance of the experience of the team members, it will be complicated to standardize such a competition. Instead, we believe that it is worth investigating different formats. As the tools will develop further, also verification competitions will develop further.

Acknowledgements. The competition received generous supported from COST Action IC0701. Huisman and Mostowski are partially supported by ERC grant 258405 for the VerCors project. Mostowski is partially supported by Artemis grant 2008-100039 for the CHARTER project.

References

1. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
2. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)
3. Bobot, F., Filliâtre, J.-C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages, Wrocław, Poland (August 2011)
4. Brockschmidt, M., Otto, C., Giesl, J.: Modular termination proofs of recursive Java Bytecode programs by term rewriting. In: Proc. RTA 2011. LIPIcs, vol. 10, pp. 155–170 (2011)
5. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
6. Distefano, D., Parkinson, M.J.: jStar: towards practical verification for Java. In: Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications, OOPSLA 2008, pp. 213–226. ACM, New York (2008)
7. Filliâtre, J.-C., Marché, C.: The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 173–177. Springer, Heidelberg (2007)
8. Fuhs, C., Giesl, J., Plücker, M., Schneider-Kamp, P., Falke, S.: Proving Termination of Integer Term Rewriting. In: Treinen, R. (ed.) RTA 2009. LNCS, vol. 5595, pp. 32–47. Springer, Heidelberg (2009)
9. Giesl, J., Schneider-Kamp, P., Thiemann, R.: AProVE 1.2: Automatic Termination Proofs in the Dependency Pair Framework. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 281–286. Springer, Heidelberg (2006)
10. Kaldewaij, A.: Programming: the derivation of algorithms. Prentice-Hall, Inc. (1990)
11. Kassios, I.T.: Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions. In: Misra, J., Nipkow, T., Karakostas, G. (eds.) FM 2006. LNCS, vol. 4085, pp. 268–283. Springer, Heidelberg (2006)
12. 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: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 154–168. Springer, Heidelberg (2011)
13. Leino, K.R.M.: Dafny: An Automatic Program Verifier for Functional Correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16 2010. LNCS, vol. 6355, pp. 348–370. Springer, Heidelberg (2010)
14. Moy, Y., Marché, C.: The Jessie plugin for Deduction Verification in Frama-C — Tutorial and Reference Manual. INRIA & LRI (2011), http://krakatoa.lri.fr/

15. Reif, W., Schellhorn, G., Stenzel, K., Balser, M.: Structured specifications and interactive proofs with KIV. In: Bibel, W., Schmitt, P. (eds.) Automated Deduction—A Basis for Applications, vol. II.1, pp. 13–39. Kluwer (1998)
16. Schmitt, P.H., Ulbrich, M., Weiß, B.: Dynamic Frames in Java Dynamic Logic. In: Beckert, B., Marché, C. (eds.) FoVeOOS 2010. LNCS, vol. 6528, pp. 138–152. Springer, Heidelberg (2011)
17. Stenzel, K.: A Formally Verified Calculus for Full Java Card. In: Rattray, C., Maharaj, S., Shankland, C. (eds.) AMAST 2004. LNCS, vol. 3116, pp. 491–505. Springer, Heidelberg (2004)