Using a Software Testing Technique to Improve Theorem Proving

Reiner Hähnle and Angela Wallenburg

Chalmers University of Technology and Göteborg University
Department of Computing Science
SE-412 96 Göteborg, Sweden
{angelaw,reiner}@cs.chalmers.se

Abstract. Most efforts to combine formal methods and software testing go in the direction of exploiting formal methods to solve testing problems, most commonly test case generation. Here we take the reverse viewpoint and show how the technique of partition testing can be used to improve a formal proof technique (induction for correctness of loops). We first compute a partition of the domain of the induction variable, based on the branch predicates in the program code of the loop we wish to prove. Based on this partition we derive a partitioned induction rule, which is (hopefully) easier to use than the standard induction rule. In particular, with an induction rule that is tailored to the program to be verified, less user interaction can be expected to be required in the proof. We demonstrate with a number of examples the practical efficiency of our method.

1 Introduction

Testing and formal verifi cation at fi rst glance seem to be at opposing ends in the spectrum of techniques for software quality assurance. Testing is a core technique used by practitioners every day, while formal verifi cation is diffi cult to master, and employed mostly by specialists in academia. Most practitioners agree that formal verifi cation is too cumbersome and diffi cult to be useful in practice. On the other hand, testing cannot be used on its own to prove the absence of errors, because exhaustive testing is usually impossible. In practice, one stops testing once the number of found errors drops below a certain threshold (or simply when the testing budget is used up). Formal verifi cation, although costly, can ensure that a program meets its (formal) specifi cation for any input. Given this state of affairs, it might seem surprising that testing and verifi cation can fruitfully interact—nevertheless, this is what we want to show in the present paper.

There is one fairly established connection between formal methods and testing (documented, for example, in several papers collected in this proceedings): test case generation from formal specifi cations. The presence of a formal specifi cation can also solve the oracle problem. One obstacle of this approach is that availability of a formal specifi cation is the exception rather than the rule. On the other hand, if the cost for providing a formal specifi cation has been invested already, one can use it as a basis not only for testing, but even for formal source code verifi cation. The contribution of this paper is to show that techniques from testing can considerably simplify the verifi cation effort. Hence, the availability of a formal specifi cation is doubly useful: on the one hand, with by now established techniques one can generate test cases automatically. In addition, as we show below, by employing techniques from testing, even formal verification may come into reach. We see our work as a first step towards a framework, where both testing and verification can be usefully combined.

Partition testing is a software testing technique used to systematically reduce test volume. A program's possibly infinite input space is divided into a finite number of disjoint subdomains. Testing is done by picking one or more elements from each subdomain to form a test set that is somehow representative for the program behaviour. Ideally, all elements in a subdomain behave in the same way with respect to the specification, that is, they are all processed correctly or they are all processed incorrectly. Subdomains with this property are called *revealing* [1] or *homogeneous* [2].

There is a line of work in software testing theory [3, 4, 1, 5, 2], where it is shown that testing *can* be used to show the *absence* of errors *provided that* certain properties in the test case selection are fullfilled. In the context of partition testing, the sought-after property is that subdomains are revealing. Unfortunately, establishing this property in practice means usually to give a formal correctness proof (for each subdomain). Hence, given the difficulties of general theorem proving, this was often discarded as impractical. Our results may be considered as a step towards obtaining such correctness proofs practically, because it suggests that proving correctness for each subdomain separately requires less user interaction than giving a proof simulteanously for the entire domain (as usually done in theorem proving).

In a nutshell, here is what we do: the implementation basis for our work is a software verification system for the programming language JAVA CARD called KeY [6]. The verification paradigm of KeY is to execute programs with symbolic values, which then are checked (symbolically) against the formal specification. More exactly, KeY is based on a first order dynamic logic with arithmetic [7, 8]. It uses a sound and relatively complete calculus which contains rules *mimicking* symbolic execution. This idea was first presented in [9] and formalized in [10, 11].

The main obstacle in automating software verification to an acceptable degree is the handling of programs with loops or recursive methods. These constructs require induction on one of the inductive data structures occurring in the program (for example, numbers or lists). The difficulty is to find a suitable induction hypothesis. This can be a formidable challenge even for formal methods experts. The complexity of the induction, of course, depends on the complexity of the loop or method body and post condition at hand. In simple cases, the induction can be performed automatically. Therefore, it would be extremely beneficial to simplify the required induction hypotheses. The key insight that we work out in the present paper is that the technique of partition testing is in fact a fairly general and *automatic* divide-and-conquer concept that can be used to simplify inductions in formal verification proofs.

Roughly speaking, to verify a loop, we use a white-box partition analysis based on the branch predicates of its body and condition, to compute a partition of the domain of the induction variable. This partition is then used to derive (mechanically) an induction rule which takes the partition into account: let us call the *standard induction rule* for natural numbers the rule that allows to conclude that a statement $\phi(n)$ holds for all $n \in \mathbb{N}$ provided that it holds for the single base case ("$\phi(0)$") and for the single step case ("for any i, if $\phi(i)$ then $\phi(i+1)$"). This is replaced by an induction rule that has m base cases and r step cases, each of which matches a subdomain of the partition and, hopefully, needs much less user interaction.

Other work that is related to ours can be found elsewhere. For instance, there was an early effort [12] to use test data to aid in proving program correctness. In contrast to this approach, we do not actually run any tests, but our approach relies on a test case generation technique (partition analysis). Also, there is a recent runtime analysis technique to generate invariants inductively from test cases, presented in [13]. For higher order functional programming languages, [14] describes how to formally derive induction schema for recursively defined functions. However, our work has the advantage to be applicable to a real object-oriented programming language, JAVA CARD.

The remainder of the paper is organized as follows. We start with a motivating example in Sect. 2. In Sect. 3 we describe the method. Then we show it at work. First, we revisit the introductory example (Sect. 4.1), followed by a more sophisticated problem (Sect. 4.2). We close by pointing out current limitations (and, hence, future work).

2 Motivating Example

In this section we describe a simple example of a loop that is not possible to prove (without complex user interaction) using a standard induction rule, but is easy with our approach. The description here is brief. Our method is explained in detailed in the following section. Here is the JAVA CARD code of the loop:

```
int final c = ... ;
int i;
...

while(i > 0) {
  if (i >= c) {
    i = i - c;
  } else {
    i--;
  }
}
```

For this while-loop to terminate in a state where $i = 0$ we need in the precondition that $i \geq 0$ and $c \geq 1$. c is constant. In dynamic logic (briefly DL—the essentials of our logical framework are described in Sect. 4.1) the proof obligation is $\forall i \cdot \phi(i)$, where $\phi(i)$ is:

```
i ≥ 0 ∧ c ≥ 1 →
  ⟨ while (i > 0) {
      if (i >= c) {
        i = i − c;
      } else {
        i−−;
      }
    } ⟩ i = 0
```

The formula contains a total correctness assertion: the program within the brackets $\langle\ \rangle$ (here the code of the while-loop) terminates and in the final state the postcondition following the brackets must hold (here $i = 0$).

The simplest possible choice for the induction hypothesis when proving correctness of the loop is to take $\phi(n)$. It is completely schematic and requires no interaction with the user. This hypothesis, however, is too weak when using the standard induction rule. Roughly speaking, in a proof attempt of the standard step case, $\forall n \in \mathbb{N} \cdot \phi(n) \rightarrow \phi(n+1)$, the following happens: the while-loop is unwound for $n+1$ and the proof branches at the if-statement. One case (the one with "`i--;`") is possible to prove, because "`(n+1)--;`" is equal to n after symbolic execution. The proof obligation for this case simplifies to $\forall n \in \mathbb{N} \cdot \phi(n) \wedge n < c \rightarrow \phi(n)$, which is valid. In the other case symbolic execution gives $n + 1 - c$ so that the resulting proof obligation $\forall n \in \mathbb{N} \cdot \phi(n) \wedge n \geq c \rightarrow \phi(n+1-c)$ is in general unprovable. With standard induction, a more powerful induction hypothesis must to be found—a difficult task for a user with no training in formal methods!

In our approach we instead create *mechanically* a new, partitioned induction rule. For our example loop the partitioned induction rule has *two* base cases and one step case:

$$\phi(0) \tag{1}$$

$$\phi(1) \wedge \cdots \wedge \phi(c-1) \tag{2}$$

$$\forall n \in \mathbb{N} \cdot \phi(n) \rightarrow \phi(n+c) \tag{3}$$

These are constructed from a branch coverage partition of the induction variable i. For instance (2) above corresponds to the subdomain with all values of i causing the "`else`" branch inside the loop to be executed. The creation of the partitioned induction rule for this particular example is described in more detail in Sect. 4.1. Note that this partitioned induction rule is powerful enough to make the proof go through automatically with the unchanged induction hypothesis $\phi(n)$ that is just what we desire in our effort to minimise the user interaction.

3 Computing Partitioned Induction Rules

The idea is to create for each loop that we want to prove a new, tailor-made induction rule based on partitions. A partition is used, through the new induction rule, to divide the proof into smaller and hopefully simpler (in terms of user interaction) parts. Here is an overview of the method:

1. Compute a partition based on the branch predicates in the program code. We employ techniques readily available in the software testing community. Details are out of the scope of this paper, but the approach we use here is similar to the construction of the implementation partition in [5].
2. Refine this partition, thereby making use of the implicit case distinction contained in operators (such as mod or $\div$) that occur in branch predicates. The goal of the partition refinement is to arrive at subdomains of a syntactic form that is suitable for generation of the new induction rule.

3. Based on the refined partition, create a new (program-specific) induction rule with one base case for each finite subdomain of the partition and one step case for each infinite subdomain.
4. Prove correctness of the loop as usual, but use the new induction rule. This requires typically less user interaction than with the standard induction rule.

Now to the details. Specifically, assume that we have a program loop with input domain, or in our case, a domain of the variable that we want to perform induction over: $D \subseteq \mathbb{N}$. From a partition analysis as described in step 1 above we obtain a finite number of disjoint subdomains, say, $D = D_1 \cup \cdots \cup D_m$. Let d_i be the characteristic predicate for each $i \in 1, \ldots, m$ with $x \in D_i$ iff $d_i(x)$ holds. Hence, $x \in D$ iff $d_1(x) \vee \cdots \vee d_m(x)$. The d_i are called *branch predicates*.

The branch predicates originate from the branching conditions in the program code and might contain operators defined by case distinction, for instance, $\div$, mod, and $\geq$. These implicit case distinctions drive further partitioning.

For each such operator, if necessary, we create a partition such that each case distinction in the definition of the operator gives rise to a new subdomain. In the future, we plan to create a library of partitions for all operators that occur in JAVA CARDexpressions and the standard JAVA CARD API so that refining partitions can be looked up mechanically. In general, we strive to refine the original partition to obtain new subdomains of a particular *syntactic form*:

1. $\{\}$ (the empty set)
2. A finite set $\{x_1, \ldots, x_k\}$. Such a set is important to distinguish because it can quite simply be used as a base case in the new induction rule.
3. An infinite set of the form $\{\lambda x.f(x) \,|\, x \in C\}$, where $C \subseteq \mathbb{N}$. It is important that $f(x)$ always increases its argument, because it is eventually to be used as an induction step in our new induction rule. We use $\lambda x.f(x)$ because we aim for an *expression* (and not the value of a function) to describe a set of values that we want to perform induction over.

Here is an example of a partition refinement based on an operator definition by case distinction: in the example from Sect. 2 one of the branch conditions contains the operator $\geq$, which has an implicit case distinction. We use the following definition of $\geq$:

$$x \geq z = \begin{cases} \mathit{true} & \text{if } \exists y \in \mathbb{N} \cdot (x = z + y) \\ \mathit{false} & \text{if } \exists y \in \mathbb{N} \cdot (x = z - 1 - y) \end{cases}$$

Each case gives directly a simple expression of the desired form: $\lambda n.(c + n)$, respectively, $\lambda n.(c - 1 - n)$ would be used to refine a subdomain defined by the predicate $i \geq c$.

More precisely, we refine the subdomain of the original partition ($\{i \in \mathbb{N} \,|\, i \geq c\}$) into two new subdomains by replacing i with $(c + n)$ and $(c - 1 - n)$ respectively. We then get $\{c - 1 - n \,|\, n \in \mathbb{N} \wedge\ c - 1 - n \geq c\} = \{\}$ and $\{c + n \,|\, n \in \mathbb{N} \wedge\ c + n \geq c\} = \{c + n \,|\, n \in \mathbb{N}\}$, which are both of the form required above. The latter can be used to derive an induction step case.

Assume now that we have a refined partition of the syntactic form detailed above, where operators with implicit case distinctions are eliminated.

We create a new induction rule with the following set of proof obligations:

1. For each non-empty finite subdomain $\{x_1, \dots, x_k\}$, we create a base case consisting of the proof obligation

$$\phi(x_1) \wedge \cdots \wedge \phi(x_k)$$

2. For each infinite subdomain D_i a new step case needs to be proven:

$$\forall n \in C_i \cdot \phi(n) \rightarrow \phi((\lambda x.f_i(x))n)$$

For the new induction rule to be sound it is important that some criteria are fulfilled:

1. For each step case of the form $\forall n \in C_i \cdot \phi(n) \rightarrow \phi((\lambda x.f_i(x))n)$ the following holds:

$$\forall n \in C_i \cdot f_i(n) > n$$

   This is to ensure that it really is a step, and it is achieved by constructing $f_i(x)$ such that it increases its argument.
2. Each element of the domain D of the induction variable is covered in at least one of the step or base cases. Let B be the union of all finite subdomains giving rise to a base case and let $f_1, \dots, f_r$ be the functions that define the step cases. Then we require

$$\forall x \in D \cdot (\exists k \in \{1, \dots, r\} \cdot \exists y \in C_k \cdot x = f_k(y)) \vee x \in B$$

   This property is guaranteed by construction, because the partition property is invariant in the process; we only refine partitions or do not change them at all.

The first property entails that the minimal element of D cannot be in the subdomain defined by any step case. The second property says that all elements of D is in either a step case or a base case. As a consequence, there must be at least one base case of the induction.

4 Examples

4.1 Simple Example Revisited

Now we return to the motivating example from Sect. 2 and show how we actually computed the partitioned induction rule for it.

In KeY, the logical infrastructure is JAVA CARD DL [8], an extension of dynamic logic (DL) [7] to handle side effects, aliasing, exceptions and other complications of a real object-oriented programming language such as JAVA CARD. In DL, a formula $\varphi \rightarrow \langle \mathtt{p} \rangle \, \psi$ is valid if for every state s satisfying precondition φ a run of the program p starting in s terminates, and in the terminating state the post-condition ψ holds. The proof obligation from Sect. 2, that was written using pure DL, is slightly more complicated in JAVA CARD DL. Our proof obligation is $\forall \, i_l \cdot \phi(i_l)$. Let $\phi(i_l)$ be the following formula:

```
$i_l \geq 0 \land c_l \geq 1 \rightarrow$
   $\{$i := $i_l\}\{$c := $c_l\}$ $\langle$ **while** (i > 0) {
                                       **if** (i >= c) {
                                          i = i − c;
                                       } **else** {
                                          i−−;
                                       }
                                    } $\rangle$ i = 0
```

The curly brackets in front of the formula are called (state) *updates*. Updates are the JAVA CARD DL solution to deal with aliasing and assignment in the calculus. They are basically primitive assignments of the form $\{loc := val\}$ where val must be a logical (side effect free) term and loc a program variable.

To prove correctness of the loop we need to perform induction on the variable i. In JAVA CARD DL one cannot quantify over program variables, so the induction variable is a corresponding logical variable i_l. The domain of the induction variable is $\mathbb{N}$. From the branching conditions in the program we obtain the first partition of i's domain:

$$\begin{aligned} D_1 &= \{x \in \mathbb{N} \,|\, d_1(x)\} = \{x \in \mathbb{N} \,|\, x \leq 0\} = \\ &= \{0\} \\ D_2 &= \{x \in \mathbb{N} \,|\, d_2(x)\} = \{x \in \mathbb{N} \,|\, x > 0 \land x < c\} = \\ &= \{1, \ldots, c-1\} \\ D_3 &= \{x \in \mathbb{N} \,|\, d_3(x)\} = \{x \in \mathbb{N} \,|\, x > 0 \land x \geq c\} = \\ &= \{x \in \mathbb{N} \,|\, x \geq c\} \end{aligned}$$

The subdomains $D_1 = \{0\}$ and $D_2 = \{1, \ldots, c-1\}$ are finite and thus already in one of our desired formats.

Then, to refine/rewrite the original subdomain D_3, remember from Sect. 3 that for the operator $x \geq z$, we may use the expressions $\lambda y.z + y$ and $\lambda y.z - 1 - y$ to refine a subdomain. This gives a refinement of $D_3 = \{x \in \mathbb{N} \,|\, x \geq c\}$ into two new subdomains $D_3 = D_{31} \cup D_{32}$, where

$$\begin{aligned} D_{31} &= (replace\ x\ in\ D_3\ with\ c + y) \\ &= \{c + y \,|\, y \in \mathbb{N} \land\ c + y \geq c\} \\ &= \{c + y \,|\, y \in \mathbb{N}\} \\ D_{32} &= (replace\ x\ in\ D_3\ with\ c - 1 - y) \\ &= \{c - 1 - y \,|\, y \in \mathbb{N} \land\ c - 1 - y \geq c\} \\ &= \{\} \end{aligned}$$

So the new subdomains D_1, D_2 and D_{31} are of the form we need to construct the new induction rule. To prove $\forall n \in \mathbb{N} \cdot \phi(n)$, it is then enough to prove

$$\phi(0) \tag{1}$$

$$\phi(1) \land \cdots \land \phi(c-1) \tag{2}$$

$$\forall n \in \mathbb{N} \cdot \phi(n) \rightarrow \phi(c+n) \tag{3}$$

where (1) is a base case and covers D_1, (2) is also a base case and covers D_2, and (3) is a step case that covers all elements in the subdomain D_{31}.

The proving process in KeY is partially automated, though it is an interactive theorem prover. When using the partitioned induction rule above, the following kinds of user interaction are required to complete the proof:

Instantiation means a single quantifier elimination by supplying a suitable instance term. In the KeY system, the user can simply drag-and-drop the desired term.

Induction rule application: when applying the partitioned induction rule, one can state the induction hypothesis by drag-and-dropping the existing proof obligation, and then pick the induction variable.

Unwinding of the loop needs to be initiated, but is done automatically.

Decision procedure is an automatic procedure that tries to decide the validity of arithmetic expressions over the integers. The decision procedure is sound but not complete. The user decides when (if) to run it.

Compared to the user interaction needed when using standard induction, this is less complicated. Using standard induction, if one uses the unmodified induction hypothesis and the same induction variable as above, one is left with an open proof goal and no rules to apply: one has to figure out a strong enough induction hypothesis.

4.2 Russian Multiplication Example

Let us see how the method works for proving the correctness of a more complicated algorithm—russian multiplication. The loop has more complicated control flow than in the previous example.

```
int russianMultiplication (int a, int b) {
   int z = 0;
   while (a != 0) {
      if (a mod 2 != 0) {
          z = z + b;
      }
      a = a/2;
      b = b*2;
   }
   return z;
}
```

For this loop we have the precondition $a_0 \geq 0$ and the post-condition $z = z_0 + a_0 * b_0$, where a_0, b_0, z_0 are the values of a, b and z before the loop. In JAVA CARD DL the proof obligation for the total correctness of this loop is $\forall a_0 \cdot \phi(a_0)$, where $\phi(a_0)$ is

```
∀b₀ · ∀z₀ · a₀ ≥ 0 →
  {a := a₀}{b := b₀}{z := z₀}⟨ while (a ! = 0) {
                                  if (a mod 2 ! = 0) {
                                     z = z + b;
                                  }
                                   a = a / 2;
                                   b = b * 2;
                               } ⟩ z = z₀ + a₀ * b₀
```

where a_0, b_0 and z_0 are new logical variables.

This cannot be proven using standard induction unless the induction hypothesis is strengthened in a non-trivial way. In an attempt to prove the standard step case using $\phi(a)$ as the induction hypothesis in $\forall a \in \mathbb{N} \cdot \phi(a) \rightarrow \phi(a+1)$, after unwinding and symbolically executing the loop we end up with $\forall a \in \mathbb{N} \cdot \phi(a) \rightarrow \phi((a+1)/2)$ which is unprovable without induction.

Now let us compute a partitioned induction rule for this loop instead. The induction variable is a (the corresponding logical variable is a_0). Its domain is $\mathbb{N}$ and the first partitioning, using branch predicates, gives the following subdomains:

$$\begin{aligned}
D_1 &= \{x \in \mathbb{N} \,|\, d_1(x)\} = \{x \in \mathbb{N} \,|\, x = 0\} \\
&= \{0\} \\
D_2 &= \{x \in \mathbb{N} \,|\, d_2(x)\} \\
&= \{x \in \mathbb{N} \,|\, x \neq 0 \wedge x \bmod 2 \neq 0\} \\
D_3 &= \{x \in \mathbb{N} \,|\, d_3(x)\} \\
&= \{x \in \mathbb{N} \,|\, x \neq 0 \wedge x \bmod 2 = 0\}
\end{aligned}$$

In words, we have the singleton set containing zero and the sets with the odd and (non-zero) even numbers respectively.

Consider the branch predicate $d_3(x) \leftrightarrow x \neq 0 \wedge x \bmod 2 = 0$, which defines subdomain D_3. The definition of $d_3(x)$ contains an operator with implicit case distinction: $\bmod$. We look up the definition of $\bmod 2$:

$$x \bmod 2 = \begin{cases} 0 \text{ if } \exists y \in \mathbb{N} \cdot (x = 2 * y) \\ 1 \text{ if } \exists y \in \mathbb{N} \cdot (x = 2 * y + 1) \end{cases}$$

Hence, we use the expressions $\lambda y.\, 2 * y$ and $\lambda y.\, 2 * y + 1$ to refine the original partition. Using the case distinction in the definition of $x \bmod 2$, gives us the refinement of the original subdomain $D_3 = \{x \in \mathbb{N} \,|\, x \neq 0 \wedge x \bmod 2 = 0\}$ into two new subdomains $D_3 = D_{31} \cup D_{32}$:

$$\begin{aligned}
D_{31} &= (\mathit{replace}\ x\ \mathit{in}\ D_3\ \mathit{with}\ 2 * y) \\
&= \{2 * y \,|\, y \in \mathbb{N} \wedge (2 * y) \neq 0 \wedge (2 * y) \bmod 2 = 0\} \\
&= \{2 * y \,|\, y \in \mathbb{N} \wedge y \neq 0 \wedge 0 = 0 \\
&= \{2 * y \,|\, y \in \mathbb{N}_1\}
\end{aligned}$$

$$\begin{aligned} D_{32} &= (replace\ x\ in\ D_3\ with\ 2*y+1) \\ &= \{2*y+1 \mid y \in \mathbb{N} \wedge 2*y+1 \neq 0 \wedge (2*y+1) \bmod 2 = 0\} \\ &= \{2*y+1 \mid y \in \mathbb{N} \wedge 1 = 0\} \\ &= \{\} \end{aligned}$$

Similarly, for the branch predicate $d_2(x)$ of the original partition, we get

$$\begin{aligned} D_{21} &= \{2*y+1 \mid y \in \mathbb{N}\} \\ D_{22} &= \{\} \end{aligned}$$

After refinement, we have non-empty subdomains of the form:

$$\begin{aligned} D_1 &= \{0\} \\ D_{21} &= \{2*y+1 \mid y \in \mathbb{N}\} \\ D_{31} &= \{2*y \mid y \in \mathbb{N}_1\} \end{aligned}$$

Thus, the new subdomains have the syntactic form we need to construct an induction rule. With this rule, to prove $\forall n \in \mathbb{N} \cdot \phi(n)$, it is enough to prove that

$$\phi(0) \tag{1}$$
$$\forall n \in \mathbb{N}_1 \cdot \phi(n) \rightarrow \phi(2*n) \tag{2}$$
$$\forall n \in \mathbb{N} \cdot \phi(n) \rightarrow \phi(2*n+1) \tag{3}$$

where (1) is the base case, covering D_1, (2) the first step case, covering all elements in the subdomain D_{31} and (3) the second step case, covering D_{21}.

To prove the russian multiplication algorithm in KeY with the partitioned induction rule the required user interactions are basically the same as in the previous example. In particular, the induction now goes through completely unmodified.

5 Limitations and Future Work

In this paper we demonstrated that the technique of partition testing can be turned into a divide-and-conquer concept to simplify inductions in formal verification proofs. We have defined a syntactic framework that allows us to derive tailor-made induction rules based on partitions in a practically efficient manner. Resulting induction rules are sound and complete by construction. The actual verification in KeY using the partitioned induction rules can often be performed automatically. Several examples were carried out not just by hand, but as concrete experiments in an interactive theorem prover. The experimental findings confirmed our conjecture. We think that our work is a first step towards a framework, where both testing and formal verification can be usefully combined.

In the current setting, our method has a number of limitations but its reach could be extended considerably. For a start, we considered induction not over arbitrary inductive data structures, but only the natural numbers. Future work is to extend our approach to also include induction over lists, trees, etc.

Our focus has been entirely on the verification of loops, and not on arbitrary programs. Since loops are usually the major source of complexity in verification, in testing as well as in theorem proving, it is here that we expect the largest gain. Still, we also wish to investigate the idea of partitioning proofs for loop-free programs, since it has been seen [15] that in the case of very large proof obligations, it is beneficial to split the proof into parts which can be handled separately.

Clearly, not all induction proofs can be simplified with our approach. The crucial point is that our method requires that the branch predicates somehow capture what is being computed in the corresponding branch. This is often the case, but not always. If the branch predicates are completely unrelated to the induction variable, we simply get no information from the branch predicates on how to partition the domain of the induction variable. For instance in array-sorting algorithms, it is common that the induction goes over the indexes of the array, but the branch predicates typically have a comparison between the elements to be sorted and these might be randomly ordered. In future work we plan to remedy this by also using the weakest preconditions of updates to induction variables when we refine the partition. In KeY there is already a strongest postcondition generator.

Finally, the process of transforming general branch predicates into predicates of the form that our method requires (using λ-expressions) is non-trivial; in particular for the process to be mechanised. In our examples, quite simple branch conditions occurred. It is future work to investigate what exactly can be done mechanically. It includes dealing with predicates containing arbitrary linear operators and method calls, but quadratic operators and operators like $\sin$, we expect to be beyond reach. However, our method is conservative in the sense that if it does not find a useful refinement of a partition for a certain subdomain, the subdomain stays the same. In that case the proof will not be simplified, but it will not be more complicated either.

Acknowledgements

We thank the anonymous reviewers and the editors for their many helpful comments. This work was also supported by a STINT (the Swedish Foundation for International Cooperation in Research and Higher Education) grant.

References

1. Weyuker, E.J., Ostrand, T.J.: Theories of Program Testing and the Application of Revealing Subdomains. IEEE Transactions on Software Engineering **6** (1980) 236–246
2. Hamlet, D., Taylor, R.: Partition Testing Does Not Inspire Confidence. IEEE Transactions on Software Engineering **16** (1990) 1402–1411
3. Goodenough, J.B., Gerhart, S.L.: Toward a Theory of Test Data Selection. IEEE Transactions on Software Engineering **1** (1975) 156–173
4. Howden, W.E.: Reliability of the Path Analysis Testing Strategy. IEEE Transactions on Software Engineering **2** (1976) 208–215
5. Richardson, D., Clarke, L.: Partition Analysis: A Method Combining Testing and Verification. IEEE Transactions on Software Engineering **11** (1985) 1477–1490

6. Ahrendt, W., Baar, T., Beckert, B., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Schmitt, P.H.: The KeY system: Integrating object-oriented design and formal methods. In Kutsche, R.D., Weber, H., eds.: Fundamental Approaches to Software Engineering. Volume 2306 of LNCS., Springer-Verlag (2002) 327–330
7. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press (2000)
8. Beckert, B.: A dynamic logic for the formal verification of Java Card programs. In Attali, I., Jensen, T., eds.: Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, Cannes, France. Volume 2041 of LNCS., Springer-Verlag (2001) 6–24
9. Burstall, R.M.: Program proving as hand simulation with a little induction. In Rosenfeld, J.L., ed.: Information Processing 74, Proceedings of IFIP Congress 74. (1974) 201–210
10. Hähnle, R., Heisel, M., Reif, W., Stephan, W.: An interactive verification system based on dynamic logic. In Siekmann, J., ed.: Proc. 8th Conference on Automated Deduction CADE, Oxford. Volume 230 of LNCS., Springer-Verlag (1986) 306–315
11. Heisel, M., Reif, W., Stephan, W.: Program verification by symbolic execution and induction. In: Proc. 11th German Workshop on Artifical Intelligence. Volume 152 of Informatik Fachberichte., Springer-Verlag (1987) 201–210
12. Geller, M.M.: Test data as an aid in proving program correctness. Communications of the ACM **21** (1978) 368–375
13. Nimmer, J.W., Ernst, M.D.: Automatic generation of program specifications. In: Proceedings of the international symposium on Software testing and analysis, ACM Press (2002) 229–239
14. Slind, K.: Derivation and use of induction schemes in higher-order logic. In Gunter, E.L., Felty, A., eds.: Proc. 10th International Theorem Proving in Higher Order Logics Conference. Volume 1275 of LNCS., Springer-Verlag (1997) 275–290
15. Claessen, K., Hähnle, R., Mårtensson, J.: Verification of hardware systems with first-order logic. In Sutcliffe, G., Pelletier, J., Suttner, C., eds.: Proc. Problems and Problem Sets Workshop, affiliated to CADE-18, Copenhagen, DIKU, University of Copenhagen, Denmark (2002) Technical Report.