Verification by Parallelization of Parametric Code*

Tobias Gedell and Reiner Hähnle

Department of Computer Science and Engineering
Chalmers University of Technology
412 96 Göteborg, Sweden
{gedell,reiner}@chalmers.se

Abstract. Loops and other unbound control structures constitute a major bottleneck in formal software verification, because correctness proofs over such control structures generally require user interaction: typically, induction hypotheses or invariants must be found or modified by hand. Such interaction involves expert knowledge of the underlying calculus and proof engine. We show that one can replace interactive proof techniques, such as induction, with automated first-order reasoning in order to deal with parallelizable loops. A loop can be parallelized, whenever the execution of a generic iteration of its body depends only on the step parameter and not on other iterations. We use a symbolic dependence analysis that ensures parallelizability. This guarantees soundness of a proof rule that transforms a loop into a universally quantified update of the state change information effected by the loop body. This rule makes it possible to employ automatic first-order reasoning techniques to deal with loops. The method has been implemented in the KeY verification tool. We evaluated its applicability with representative case studies from the JAVA CARD domain.

1 Introduction

The context of this paper is formal software verification of object-oriented programs. The target programs are executable JAVA programs (not abstract programs) and we want to prove complex functional properties of these. There are a number of software verification systems that target JAVA and related programming languages [2, 4, 10, 18, 22, 26]. All of these systems are semi-automatic at best. The reason is that the emergence of undecidable predicates is typical when proving correctness for the combination of data structures of unbounded size and of control structures that can lead to an unbounded number of execution steps. Typical examples of the former include integers, lists (arrays), trees. The most important representatives of the latter are loops, recursive method calls, and concurrent processes. All of them are present in JAVA-like languages.

If we do not want to abstract away from real JAVA programs as in software model checking [15] or trade off verification for mere bug finding [12], then the inherent lim-

* This work was funded in part by a STINT institutional grant and by the IST programme of the EC, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project. This article reflects only the author's views and the Community is not liable for any use that may be made of the information contained therein. This paper is an extended and revised version of [13].

itations of computability do not allow a complete, uniform deduction system for program verification. Even though it seems that the calculi used for program verification are practically complete in the sense that complex, realistic examples can be handled [4, 8, 16] without encountering incompleteness phenomena, this is not enough. To see why, let us look at an example.

Example 1 (Array reversal).
The following loop reverses the elements of the **int** array `a`:

```
int half = a.length / 2 - 1;
for (int i = 0; i <= half; i++) {
  int tmp = a[i];
  a[i] = a[a.length - 1 - i];
  a[a.length - 1 - i] = tmp;
}
```

A formal specification can be given in first-order logic as follows:

Precondition: $\texttt{a} \not\doteq \textbf{null}$
Postcondition: $\forall j.(0 \leq j < \texttt{a.length} \rightarrow \texttt{a}[j] \doteq \backslash\texttt{old}(\texttt{a})[\texttt{a.length} - 1 - j])$

The keyword `\old` indicates that the value `a` had before the execution is referred to. □

What are the options to prove total correctness of this loop with respect to its contract? Finite unwinding is impossible and abstraction has difficulties to record that the value `a.length` depends on `a`. The standard approach is to use one of two general-purpose mechanisms for dealing with unbounded control structures, *invariants* or *induction*. In the first case, one would establish that the loop preserves a suitable invariant property I, which must be strong enough to imply the postcondition. Termination of the loop is proven separately (and is trivial for this example). Alternatively, an induction argument over `i` would typically establish that the loop reverses all array positions. The problem is that both, a suitable invariant and a suitable induction hypothesis, are not straightforward to derive from the postcondition: it is necessary to introduce a new variable k for the index up to which the array has been reversed already, k must have appropriate bounds, the precondition must be included, etc. In general, the postcondition might not be given (for example, if the task is to derive the specification from the code). In this case, it is even more difficult to come up with a suitable invariant or induction hypothesis. In addition, loop rules in realistic imperative languages [6] are very complex. User interaction involves a high amount of technical knowledge and is thus extremely expensive.

There is a large body of work on heuristically guided inductive theorem proving, but most of it is done in the context of functional programming [7, 9]. Existing work on automatic synthesis of loop invariants in imperative programs [17, 23] is defined only for an abstract while-language. A recent divide-and-conquer technique for decomposition of induction proofs [20] works for a larger fragment of JAVA, but it is targeted at simplifying user interaction rather than eliminating it.

Main Contributions

The contribution of this paper is to present a new verification technique that relies neither on abstraction, nor on invariants, nor on induction. It is *complementary* to the work cited above in so far as our goal is to recognize situations where complex invariants can be avoided altogether.

The key insight (illustrated by means of Example 1) is that the swap operations realized in the loop body can be executed independently of each other: the assignment to `a[i]` and the value of `a[a.length - 1 - i]` do not depend on any `a[j]` and `a[a.length - 1 - j]` with $\mathtt{i} \neq \mathtt{j}$ provided that `i` and `j` are within the bounds specified in the guard of the loop.

Now, a new way to prove correctness of the loop goes as follows: first compute the effect of a generic iteration of the loop body parameterized with `i`; second, prove that there are no dependencies between different iterations in the loop range; third, generalize the effect of the loop body over all values that the parameter `i` takes on in the loop range; and fourth, prove that the postcondition is implied by the loop. Importantly, the last step involves no induction, but automatable first-order search stratagems such as quantifier elimination and term rewriting.

Obviously, verification by parallelization of parametric code is an *incomplete* verification technique for loops, because not all loops are parallelizable. On the other hand, it is not an exotic special case either: from an analysis of the *unchanged* code of several real JAVA CARD programs we concluded that parallelizable loops occur naturally and relatively frequently, see Section 10. As we show in Section 11, verification by parallelization is not restricted to loops, but can be applied whenever a non-linear program is composed of parametric pieces of code, for example, in recursive calls and concurrent processes. In addition, the current trend towards multi-core processors will result in more code being written in such a way that it is parallelizable. Therefore, verification by parallelization is a *relevant* technique for increasing the degree of automation in software verification.

The most important aspect of verification by parallelization is that it is a *highly automatable* verification technique. First, because the computation of the effect (i.e., the strongest postcondition relative to a given precondition) $\mathcal{U}(\mathtt{i})$ of a piece of code `p(i)` parameterized by `i` is done automatically. Even the choice of the parameter `i` is automatic and guided by heuristics. The details are given in Sections 4 and 5. Second, the effect of some non-linear parameterized code (such as a loop with body `p(i)`) is represented in form of a universally quantified state update, say, $\backslash\mathtt{for}\ \mathbf{int}\ I;\ \{\mathtt{i}:=I\}\{\ \mathcal{U}(\mathtt{i})\}$. Therefore, it can be further processed during the remaining verification proof by employing first-order reasoning, see Section 8.

Soundness of the universal quantification step is ensured by an automatic symbolic dependence analysis described in Sections 6 and 7. This analysis is executed not directly on the code `p(i)`, but on the simplified and normalized effect $\mathcal{U}(\mathtt{i})$ computed by symbolic execution before. This feature makes our approach *robust*, because the success of the dependence analysis does not rely on any syntactic restrictions of `p(i)`. A further robustness feature is that our dependence analysis does not simply fail in case when dependencies are detected, but yields a symbolic constraint that is sufficient for dependencies not to occur and that can be used elsewhere in the verification attempt, see Section 9.

In the following section, we collect a number of technical notions needed later on. In Section 3, we walk informally through the method guided by an example. The remaining sections then give the technical details.

2 Basic Definitions

The platform for our experiments is the KeY tool [4], which features an interactive theorem prover for formal verification of sequential JAVA programs.

2.1 Dynamic Logic for JAVA CARD

In KeY the target program to be verified and its specification are both modeled in an instance of a dynamic logic (DL) [14] calculus called JAVA DL [3]. JAVA DL extends other variants of DL used for theoretical investigations or verification purposes, because it handles such phenomena as side effects, aliasing, object types, exceptions, and finite integer types. JAVA DL fully axiomatizes the JAVA CARD programming language [27] which contains all JAVA features minus multi-threading, floating point types, and dynamic class loading. It has also some features that JAVA does not have, but they are not addressed in this article.

Deduction in the JAVA DL calculus is based on symbolic program execution and simple program transformations and so is close to a programmer's understanding of JAVA. It can be seen as a modal logic with a modality $\langle \mathtt{p} \rangle$ for every program p, where $\langle \mathtt{p} \rangle$ refers to the final state (if p terminates normally) that is reached after executing p.

The *program formula* $\langle \mathtt{p} \rangle \phi$ expresses that the program p terminates in a state in which ϕ holds without throwing an exception. A formula $\phi \rightarrow \langle \mathtt{p} \rangle \psi$ is valid if for every state $\mathcal{S}$ satisfying precondition ϕ a run of the program p starting in $\mathcal{S}$ terminates normally, and in the terminating state the postcondition ψ holds.

The programs occurring in JAVA DL formulas are executable JAVA code. Each rule of the JAVA DL calculus specifies how to execute symbolically one particular statement, possibly with additional restrictions. When a loop or a recursive method call is encountered, it is in general necessary to perform induction over a suitable data structure. In this paper we show how induction can be avoided in the case of parallelizable loops.

2.2 State Updates

In JAVA (as in other object-oriented programming languages), different object type variables may refer to the same object. This phenomenon, called aliasing, causes difficulties for the handling of assignments in a calculus for JAVA DL. For example, whether or not the formula $\mathtt{o1.f} \doteq \mathtt{1}$ holds after (symbolic) execution of the assignment `o2.f = 2;`, depends on whether `o1` and `o2` refer to the same object. Therefore, JAVA assignments cannot be symbolically executed by syntactic substitution without causing excessive branching. In the JAVA DL calculus a different solution is used, based on the notion of (state) *updates*.

Definition 1. Atomic updates *are of the form* `loc`$:=$`val`*, where* `val` *is a logical term without side effects and* `loc` *is either (i) a program variable* `v`*, or (ii) a field access* `o.f`*, or (iii) an array access* `a[i]`*. Updates may appear in front of any formula, where they are surrounded by curly brackets for easy parsing. The semantics of* $\{$`loc`$:=$`val`$\}\phi$ *is the same as that of* $\langle$`loc=val;`$\rangle\,\phi$.

Changes of the computation state can be represented with the help of updates. For example, the update $\{$`loc`$:=$`val`$\}\phi$ represents all states in which the formula ϕ holds after the value of `loc` has been changed to `val`. In a somewhat loose manner we use updates to represent states, for example, the update $\{$`loc`$:=$`val`$\}$ is used to represent an arbitrary state, where the value of `loc` is `val`.

Definition 2. *General* updates *are defined inductively based on atomic updates. If $\mathcal{U}$ and $\mathcal{U}'$ are updates then so are: (i) $\mathcal{U}$, $\mathcal{U}'$ (*parallel composition*), (ii) $\mathcal{U}$; $\mathcal{U}'$ (*sequential composition*), (iii)* `\if` $(b)\ \{\mathcal{U}\}$*, where b is a quantifier-free formula (*conditional execution*), (iv)* `\for` $T\ s;\ \mathcal{U}(s)$*, where s is a variable over a well-ordered type T and $\mathcal{U}(s)$ is an update with occurrences of s (*quantification*), (v)* $\{\mathcal{U}\}\,\mathcal{U}'$ application*.*

The semantics of sequential, conditional, and application updates is obvious; the meaning of a parallel update is the simultaneous application of all its constituent updates except when two left hand sides refer to the same location: in this case the syntactically later update wins. This models natural program execution flow. The semantics of `\for` $T\ s;\ \mathcal{U}(s)$ *is the parallel execution of all updates in* $\bigcup_{x \in T}\{s:=x;\mathcal{U}(s)\}$*. As for parallel updates, a last-win clash-semantics is in place: the maximal*[1] *update with respect to the well-order on T and the syntactic order within each $\mathcal{U}(s)$ wins.*

The restriction that right-hand sides of updates must be side effect-free is not essential: by introducing fresh local variables and symbolic execution of complex expressions the JAVA DL calculus rules normalize arbitrary assignments so that they meet the restrictions of updates. A full formal treatment of updates is in [24], see also [4].

Sequential composition of updates is automatically transformed into parallel composition in KeY and we will therefore mostly not consider it further.

3 Outline of the Approach

Let us look at the following example:

```
for (int i = 1; i < a.length; i++)
    if (c != 0) a[i] = b[i+1];
    else a[i] = b[i-1];
```

In a first step, the loop initialization expression is transformed out of the loop and symbolically executed. The reason is that the initialization expression might be complex and have side effects. This results in a state $\mathcal{S} = \{$`i`$:=1\}$. The remaining loop now has the form: **for** `(; i < a.length; i++)...`

[1] Well-orders are usually defined with respect to minimal elements. We use the dual definition here, because it is more natural in our setting.

We proceed to symbolically execute the loop body, the step expression, and the guard for a generic value of `i`. In order to do this correctly, we must eliminate from the current state all locations that can potentially be modified in the body, step, or guard. In Section 4 we describe an algorithm that approximates such a set of locations rather precisely. Applied to the present example we obtain `i` and `a[i]` as modifiable locations. Consequently, generic execution of the loop body, step, and guard starts in the empty state. Note that the set of modifiable locations does not include, for example, `c`. This is important, because if $\mathcal{S}$ contains, say, $\mathtt{c} := 1$, we would start the execution in the state $\{\mathtt{c} := 1\}$ and the resulting state would be much simplified.

In our example, symbolic execution of one loop iteration starting in the empty state gives $\mathcal{S}' = \{\mathtt{i} := \mathtt{i} + 1,\ \backslash\mathtt{if}\ (c \neq 0)\ \{\mathtt{a[i]} := \mathtt{b[i+1]}\},\ \backslash\mathtt{if}\ (c \doteq 0)\ \{\mathtt{a[i]} := \mathtt{b[i-1]}\}\}$, where the step and guard expressions were executed as well.

The next step is to check whether the state update $\mathcal{S}'$ resulting from the execution of the generic iteration contains dependencies that make it impossible to represent the effect of the loop as a quantified update. For $\mathcal{S}'$ this is the case if and only if `c` is 0 and `a` and `b` are the same array. In this case, the body amounts to the statement `a[i] = a[i-1]` which contains a data dependence that cannot be parallelized. All other dependencies can be captured by parallel execution of updates with last-win clash-semantics. The details of the dependence analysis are explained in Section 6 and Section 7. In the example it results in a logical constraint $\mathcal{C}$ that, among other things, contains the disjunction of $\mathtt{c} \neq 0$ and $\mathtt{a} \neq \mathtt{b}$. A further logical constraint $\mathcal{D}$ strengthening $\mathcal{C}$ is computed which, in addition, ensures that the loop terminates normally. In the example, normal termination is ensured by `a` and `b` not being **null** and `b` having enough elements, that is, $\mathtt{b.length} > \mathtt{a.length}$.

At this point the proof is split into two cases using cut formula $\mathcal{D}$. Under the assumption $\mathcal{D}$ the loop can be transformed into a quantified update. If $\mathcal{D}$ is not provable, then the loop must be also tackled with a conventional induction rule, but one may use the additional assumption $\neg\mathcal{D}$, which may well simplify the proof.

For the sake of illustration assume now $\mathcal{S}$ and $\mathcal{S}'$ both contain $\{\mathtt{c} := 1\}$ and the termination constraint in $\mathcal{D}$ holds. In this case, we can additionally simplify $\mathcal{S}'$ to $\{\mathtt{c} := 1,\ \mathtt{i} := \mathtt{i} + 1,\ \mathtt{a[i]} := \mathtt{b[i+1]}\}$.

In the final step we synthesize from (i) the initial state $\mathcal{S}$, (ii) the effect of a generic execution of an iteration $\mathcal{S}'$ and (iii) the guard, a state update, where the loop variable `i` is universally quantified. The details are explained in Section 8. The result for the example in somewhat simplified manner is as follows:

$$\begin{array}{l}\backslash\mathtt{for}\ \textbf{int}\ n;\\ \qquad \{\mathtt{i} := n + 1\}\{\backslash\mathtt{if}\ (\mathtt{i} \geq 1 \wedge \mathtt{i} < \mathtt{a.length})\ \{\mathtt{c} := 1, \mathtt{i} := \mathtt{i} + 1, \mathtt{a[i]} := \mathtt{b[i+1]}\}\}\end{array}$$

Here we make use of an update applied to an update. The variable n holds the iteration number, i.e., 0 for the first iteration, 1 for the second, and so on. For each iteration we need to assign the loop variable its value. This is done by the update $\{\mathtt{i} := n + 1\}$. We apply this update to the guarded update which has the effect that all occurrences of `i` in non-update positions (guard, arguments, right-hand sides) are replaced by $n + 1$. The resulting update is:

$\backslash$`for` **int** n;

$$\{\backslash\texttt{if}\ (n+1 \geq 1 \wedge n+1 < \texttt{a.length})\ \{\texttt{c}:=1,\ \texttt{i}:=n+2,\ \texttt{a[}n+1\texttt{]}:=\texttt{b[}n+2\texttt{]}\}\}$$

The `for`-expression is a universal first-order quantifier whose scope is an update that contains occurrences of the variable n (see Def. 2 and [24]). Subexpressions are first-order terms that are simplified eagerly while symbolic execution proceeds. first-order quantifier elimination rules based on skolemization and instantiation are applicable, for example, for any positive value j such that $j <$ `a.length` we obtain immediately the update `a[`j`]`$:=$`b[`j`+1]` by instantiation. Proof search is performed by the usual first-order strategies without user interaction.

4 Computing the Effect of a Generic Loop Iteration

In this section we describe how we compute the state modifications performed by a generic loop iteration. As a preliminary step we move the initialization out of the loop and execute it symbolically, because the initialization expression may contain side-effects. We are left with a loop consisting of a guard, a step expression, and a body:

```
for (; guard; step) body                                    (1)
```

We want to compute the state modifications performed by a generic iteration of the loop. A single loop iteration consists of executing the body, evaluating the step expression, and testing the guard expression. This behavior is captured in the following compound statement where `dummy` is needed, because JAVA expressions are not statements.

```
body; step; boolean dummy = guard;                          (2)
```

We proceed to symbolically execute the compound statement (2) for a generic value of the loop variable. This is quite similar to computing the strongest post condition of a given program. Platzer [21] has worked out the details of how to compute the strongest post condition in the specific JAVA program logic that we use and our methods are based on the same principles. Our method handles the fragment of JAVA that the symbolic execution machinery of KeY handles, which is JAVA CARD [27].

Let `p` be the code in (2). The main idea is to try to prove validity of the program formula $\mathcal{S}\langle \texttt{p}\rangle\, \mathit{fin}$, where *fin* is an arbitrary, but unspecified non-rigid predicate that signifies when to stop symbolic execution. Complete symbolic execution of `p` starting in state $\mathcal{S}$ eventually yields a proof tree whose open leaves are of the form $\Gamma \rightarrow \mathcal{U}\mathit{fin}$ for some update expression $\mathcal{U}$. The predicate *fin* cannot be shown to be true or false in the program logic. Therefore, after all instructions in `p` have been executed, symbolic execution is stuck. At this stage we extract two vectors $\vec{\Gamma}$ and $\vec{\mathcal{U}}$ consisting of corresponding Γ and $\mathcal{U}$ from all open leaf nodes. Different leaves correspond to different execution paths in the loop body.

Example 2. Consider the following statement `p`:

```
if (i > 2) a[i] = 0 else a[i] = 1; i = i + 1;
```

After the attempt to prove $\langle \texttt{p} \rangle \mathit{fin}$ becomes stuck there are two open leaves:

$$\begin{array}{l} V \wedge i > 2 \rightarrow \{\texttt{a[i]} := 0, \texttt{i} := \texttt{i} + 1\}\mathit{fin} \\ V \wedge i \not> 2 \rightarrow \{\texttt{a[i]} := 1, \texttt{i} := \texttt{i} + 1\}\mathit{fin} \end{array}$$

where V stands for $\texttt{a} \not\doteq \textbf{null} \wedge i \geq 0 \wedge \texttt{i} < \texttt{a.length}$. We extract the following vectors:

$$\begin{aligned} \vec{\Gamma} &\equiv \langle V \wedge i > 2, V \wedge i \not> 2 \rangle \\ \vec{\mathcal{U}} &\equiv \langle \{\texttt{a[i]} := 0, \texttt{i} := \texttt{i} + 1\}, \{\texttt{a[i]} := 1, \texttt{i} := \texttt{i} + 1\} \rangle \end{aligned} \tag{3}$$

□

Symbolic execution can become stuck at a leaf containing a program in three ways:

1. The program has been fully executed and only an update and the formula *fin* remain. This is what we call a *success leaf*. The effect of the program was successfully transformed into a state update. Success leaves are always of the form $\Gamma \rightarrow \mathcal{U}\mathit{fin}$.
2. Abrupt termination caused by, for example, a thrown exception. In this case the program cannot be transformed into a state update. We call this a *failed leaf*.
3. The strategies for automatic symbolic execution were not strong enough to execute all instructions in the program. This could possibly be remedied by enabling more powerful and expensive strategies and restart symbolic execution. If they are still not strong enough, we count the leaf as a failed leaf.

If a failed leaf can be reached from the initial state, then our method cannot handle the loop. We must, therefore, make sure that our method is only applied to loops for which we have proven that no failed leaf can be reached. We construct the vector $\vec{\mathcal{F}}$ consisting of the path conditions Γ of all failed leaves and let the negation of $\vec{\mathcal{F}}$ become a condition that needs to be proven when applying our method.

Example 3. In Example 2 we only showed the success leaves. When symbolic execution becomes stuck, there are, in addition to the success leaves, failed leaves of the following form:

$$\begin{array}{ll} \texttt{a} \doteq \textbf{null} & \rightarrow \ldots \mathit{fin} \\ \texttt{a} \not\doteq \textbf{null} \wedge \texttt{i} < 0 & \rightarrow \ldots \mathit{fin} \\ \texttt{a} \not\doteq \textbf{null} \wedge \texttt{i} \not< \texttt{a.length} & \rightarrow \ldots \mathit{fin} \end{array}$$

The first leaf corresponds to the case where `a` is **null** and using `a` throws a null pointer exception. The second and third leaves correspond to the case where `i` is outside `a`'s bounds and accessing `a[i]` throws an index out of bounds exception. From the failed leaves we extract the following vector:

$$\vec{\mathcal{F}} \equiv \langle \texttt{a} \doteq \textbf{null}, \texttt{a} \not\doteq \textbf{null} \wedge \texttt{i} < 0, \texttt{a} \not\doteq \textbf{null} \wedge \texttt{i} \not< \texttt{a.length} \rangle$$

□

Note that symbolic execution discards any code that cannot be reached. As a consequence, an exception that occurs at a code location that cannot be reached from the initial state will not occur in the leaves of the proof tree. This means that our method is not restricted to code that cannot throw any exception, which would be very restrictive.

So far we said nothing about the state in which we start a generic loop iteration. Choosing a suitable state requires some care, as the following example shows.

Example 4. Consider the following code:

```
c = 1;
i = 0;
for (; i < a.length; i++) {
    if (c != 0) a[i] = 0;
    b[i] = 0; }
```

At the beginning of the loop we are in state $\mathcal{S}_{\text{init}} = \{\texttt{c} := 1, \texttt{i} := 0\}$. It is tempting, but wrong, to start the generic loop iteration in this state. The reason is that `i` has a specific value, so one iteration would yield $\{\texttt{a[0]} := 0, \texttt{b[0]} := 0, \texttt{i} := 1\}$, which is the result after the *first* iteration, not a generic one. The problem is that $\mathcal{S}_{\text{init}}$ contains information that is not invariant during the loop. Starting the loop iteration in the empty state is sound, but suboptimal. In the example, we would get $\{\backslash\texttt{if}\ (\texttt{c} \neq 0)\ \{\texttt{a[i]} := 0\}$, $\texttt{b[i]} := 0,\ \texttt{i} := \texttt{i} + 1\}$, which is unnecessarily imprecise, since we know that `c` is equal to 1 during the entire execution of the loop. □

We want to use as much information as possible from the state $\mathcal{S}_{\text{init}}$ at the beginning of the loop and only remove those parts that are not invariant during all iterations of the loop. Executing the loop in the largest possible state corresponds to performing dead code elimination. When we reach a loop of the form (1) in state $\mathcal{S}_{\text{init}}$ we proceed as follows:

1. Execute **boolean** `dummy = guard;` in state $\mathcal{S}_{\text{init}}$ and obtain $\mathcal{S}$. We need to evaluate the guard since it may have side effects. Evaluation of the guard might cause the proof to branch, in which case we apply the following steps to *each* branch. If our method cannot be applied to one of the branches we backtrack to state $\mathcal{S}_{\text{init}}$ and use the standard rules to prove the loop. If the guard evaluates to false, we skip the loop and proceed using the standard rules.
2. Compute the vectors $\vec{\Gamma}$, $\vec{\mathcal{U}}$ and $\vec{\mathcal{F}}$ from (2) starting in state $\mathcal{S}$.
3. Obtain $\mathcal{S}'$ by removing from $\mathcal{S}$ all those locations that are modified in a success leaf. This is done as follows: for each modified location in $\mathcal{S}$, add an update of the location to itself in parallel to the updates in $\mathcal{S}$. They are added syntactically after all updates in $\mathcal{S}$ and, therefore, the clash-semantics of updates ensures that the previous assignments to the modified locations in $\mathcal{S}$ are canceled. More formally, $\mathcal{S}'$ is defined as follows: $\mathcal{S}' = \mathcal{S}, \bigcup_{l \in mod(\vec{\mathcal{U}}, \mathcal{S})} \{l := l\}$, where $mod(\vec{\mathcal{U}}, \mathcal{S})$ is the set of locations in $\mathcal{S}$ whose assigned term in $\vec{\mathcal{U}}$ differs from its assigned term in $\mathcal{S}$. How to compute this set is discussed below.
4. If $\mathcal{S}' = \mathcal{S}$ then stop; otherwise let $\mathcal{S}$ become $\mathcal{S}'$ and goto Step 2.

The algorithm terminates since the number of locations that can be removed from the initial state is bound both by the textual size of the loop[2] and, in case the state does not contain any quantified update, the size of the state itself. The final state of this algorithm is a greatest fixpoint containing as much information as possible from the initial state $\mathcal{S}$. Let us call this final state $\mathcal{S}_{\text{iter}}$.

Example 5. Example 4 yields the following sequence of states:

Round	Start state	State modifications	New state	Remark
1	{`c`:=`1`, `i`:=`0`}	{`a[0]`:=`0`, `b[0]`:=`0`, `i`:=`1`}	{`c`:=`1`, `i`:=`i`}[3]	
2	{`c`:=`1`, `i`:=`i`}	{`a[i]`:=`0`, `b[i]`:=`0`, `i`:=`i+1`}	{`c`:=`1`, `i`:=`i`}	Fixpoint

□

Computing the set $mod(\vec{\mathcal{U}}, \mathcal{S})$ can be difficult. Assume $\mathcal{S}$ contains `a[c]`:=0 and $\vec{\mathcal{U}}$ contains `a[i]`:=1. If `i` and `c` can have the same value then `a[c]` should be removed from $\mathcal{S}$, otherwise it is safe to keep it. In general it is undecidable whether two variables can assume the same value. A similar situation occurs when $\mathcal{S}$ contains `a.f`:=0 and $\vec{\mathcal{U}}$ contains `b.f`:=1. If `a` and `b` are references to the same object then `a.f` must be removed from the new state. These issues are handled by using a dependence analysis to compute $mod(\vec{\mathcal{U}}, \mathcal{S})$. The details of how this is done are described in Section 7.

5 Loop Variable and Loop Range

For the dependence analysis and for creating the quantified state update we need to identify a loop variable and the loop range. The requirement we have on a loop variable is that it must, in each success leaf, be updated with the same step function by an unguarded update.

When deciding whether a particular variable `i` is a possible loop variable, we look for a function *step* such that $i := step(i)$ is found in each update $\mathcal{U} \in \vec{\mathcal{U}}$. Remember that $\vec{\mathcal{U}}$ contains the updates from all success leaves. In KeY, finding such a function is often not possible due to eager simplification performed on updates. If, for example, for a specific leaf, the path condition contains $i \doteq 0$ the update $i := i + c$ will be simplified to $i := c$. This means that even if $i := i + c$ is the step function of the loop it will not be found in all leaves. To handle this we must take the path condition Γ into account. For each success leaf with path condition Γ and update $\mathcal{U}$ we require that under the path condition, $step(i)$ is equal to the expression assigned to `i` by $\mathcal{U}$. Formally, this is expressed by $\Gamma \rightarrow step(i) \doteq \mathcal{U}i$.

The step function describes the execution order of the loop iterations and expresses how the loop variable changes between each loop iteration. For constructing the quantified state update we need to know the value that the loop variable has in each iteration of the loop, that is, we need to have a function from the number of an iteration to the value of the loop variable in that iteration. This function is defined as $iter(n) = step^n(start)$ where n is the number of the iteration and *start* the initial value of the loop variable.

[2] Including the size of any method called by the loop.

[3] The new state that gets computed is {`c`:=`1`, `i`:=`0`, `i`:=`i`} but is simplified to {`c`:=`1`, `i`:=`i`}.

In JAVA DL we cannot write recursive expressions directly, so we have to rewrite the body of *iter* into a non-recursive expression. This is in general hard, but whenever the loop variable is incremented or decremented with a constant value in each iteration, it is easy to do. At present we impose this as a restriction: the step function must have the form `i + e`, where `i` is the loop variable and `e` is invariant during loop execution. Then one obtains the following definition: $iter(n) = start + n * e$. It would be possible to let the user provide the definition of *iter* allowing for more complicated step functions to be handled. It would, however, come at the price of making the method less automatic.

To identify the loop variable, we start with the set of variables occurring in the loop and remove all those for which a step function cannot be found. After this we might be left with more than one variable. Since we cannot, currently, handle more than one loop variable we need to eliminate the other candidates. If they are not eliminated they would cause data flow-dependencies that could not be handled by our method. A candidate is eliminated by transforming its expression into one which is not dependent on the candidate location. For example, the candidate `l`, introduced by the assignment `l = l + c;`, can be eliminated by transforming the assignment into `l = init + n * c;`, where `init` is the initial value of `l` and `n` the number of the iteration.

To make the identification of loop variables more efficient we use a heuristic that favors variables that occur in the loop guard (as loop variables often do) and that are syntactically small (for example, `i` is considered smaller than `a[l]`).

Example 6. Consider the code in Example 2 which gives the vector in (3). The only variable for which a step function can be found is `i`. It is, therefore, identified as the loop variable. □

To determine the loop range we begin by computing the specification of the guard in a similar way as we computed the state modifications of a generic iteration in the previous section. We attempt to prove $\langle \textbf{boolean}\ \texttt{dummy = guard;} \rangle \mathit{fin}$. From the open leaves of the form $\Gamma \rightarrow \{\texttt{dummy} := \texttt{e}, \ldots\}\mathit{fin}$, we create the formula *GS* which characterizes when the guard is true. Formally, *GS* is defined as $\bigvee_{\Gamma}(\Gamma \wedge \texttt{e} \doteq \textbf{true})$. The formula *GF* characterizes when the guard is *not* successfully evaluated. We let *GF* be the disjunction of all Γ' from the open leaves that are not of the form above.

Example 7. Consider the following guard $\texttt{g} \equiv \texttt{i < a.length}$. When all instructions in the formula $\langle \textbf{boolean}\ \texttt{dummy = g;} \rangle \mathit{fin}$ have been symbolically executed, there are two success leaves:

$$\begin{array}{l}\texttt{a} \not\doteq \textbf{null} \wedge \texttt{i} < \texttt{a.length} \rightarrow \{\texttt{dummy} := \textbf{true}\}\mathit{fin} \\ \texttt{a} \not\doteq \textbf{null} \wedge \texttt{i} \not< \texttt{a.length} \rightarrow \{\texttt{dummy} := \textbf{false}\}\mathit{fin}\end{array}$$

From these we extract the following formula *GS*:

$$\begin{array}{l}(\texttt{a} \not\doteq \textbf{null} \wedge \texttt{i} < \texttt{a.length} \wedge \textbf{true} \doteq \textbf{true}) \vee \\ (\texttt{a} \not\doteq \textbf{null} \wedge \texttt{i} \not< \texttt{a.length} \wedge \textbf{false} \doteq \textbf{true})\end{array}$$

After simplification of *GS* we obtain:

$$\texttt{a} \not\doteq \textbf{null} \wedge \texttt{i} < \texttt{a.length} \quad .$$

When the instructions have been symbolically executed, there is also a failed leaf containing $\mathtt{a} \doteq \mathbf{null} \rightarrow \ldots$ *fin*. From it we extract the formula $GF \equiv \mathtt{a} \doteq \mathbf{null}$. □

The formula GR_n characterizes when the iteration number n is within the loop range. The following definition expresses that there exists an iteration where the loop variable has the value $iter(n)$ and, moreover, this iteration can be reached:

$$GR_n \equiv n \geq 0 \ \wedge \ \{\mathtt{i} := iter(n)\}GS \ \wedge \ \forall m.\ 0 \leq m < n \rightarrow \{\mathtt{i} := iter(m)\}GS$$

It is important that the loop terminates, otherwise, our method would be unsound. We, therefore, create a termination constraint GT that needs to be proven when applying our method. The termination constraint expresses that there exists a number n of iterations after which the guard formula evaluates to false. The constraint GT is defined as:

$$GT \equiv \exists n.\ n \geq 0 \wedge \{\mathtt{i} := iter(n)\}\neg GS$$

6 Dependence Analysis

Transforming a loop into a quantified state update is only possible when the iterations of the loop are independent of each other. Two loop iterations are independent of each other if the execution of one iteration does not affect the execution of the other. According to this definition, the loop variable clearly causes dependence, because each iteration both reads its current value and updates it. We will, however, handle the loop variable by quantification. Therefore, it is removed from the update before the dependence analysis is begun. The problem of loop dependencies was intensely studied in loop vectorization and parallelization for program optimization on parallel architectures. Some of our concepts are based on results in this field [1, 28].

6.1 Classification of Dependencies

In our setting we encounter three different kinds of dependence; *data flow-dependence*, *data anti-dependence*, and *data output-dependence*.

Example 8. It is tempting to assume that it is sufficient for independence of loop iterations that the final state after executing a loop is independent of the order of execution, but the following example shows this to be wrong:

```
for (int i = 0, sum = 0; i < a.length; i++) sum += a[i];
```

The loop computes the sum of all elements in the array `a` which is independent of the order of execution, however, running all iterations in parallel gives the wrong result, because reading and writing of `sum` collide. □

Definition 3. *Let $\mathcal{S}_J$ be the final state after executing a generic loop iteration over variable `i` during which it has value J and let $<$ be the order on the type of `i`.*

There is a data input-dependence *between iterations $K \neq L$ iff $\mathcal{S}_K$ writes to a location (i.e., appears on the left-hand side of an atomic update) that is read (appears on the right hand side of an atomic update or as an argument or in a guard of an update) in $\mathcal{S}_L$. We speak of* data flow-dependence *when $K < L$ and of* data anti-dependence*, when $K > L$.*

There is data output-dependence *between iterations $K \neq L$ iff $\mathcal{S}_K$ writes to a location that is overwritten in $\mathcal{S}_L$.*

Example 9. When executing the second iteration of the following loop, the location `a[1]`, modified by the first iteration, is read, indicating data flow-dependence:

```
for (int i = 1; i < a.length; i++) a[i] = a[i - 1];
```

The following loop exhibits data output-dependence:

```
for (int i = 1; i < a.length; i++) last = a[i];
```

Each iteration assigns a new value to `last`. When the loop terminates, `last` has the value assigned to it by the last iteration. □

Loops with data flow-dependencies cannot be parallelized, because each iteration must wait for a preceding one to finish before it can perform its computation.

In the presence of data anti-dependence swapping two iterations is unsound, but parallel execution is possible provided that the generic iteration acts on the original state before loop execution begins. In our translation of loops into quantified state updates in Section 8 below, this is ensured by simultaneous execution of all updates. Thus, we can handle loops that exhibit data anti-dependence. The final state of such loops depends on the order of execution, so independence of the order of executions is not only insufficient (Example 8) but even unnecessary for parallelization.

Even loops with data output-dependence can be parallelized by assigning an ordinal to each iteration. An iteration that wants to write to a location first ensures that no iteration with higher ordinal has already written to it. This requires a total order on the iterations. From the step function we extracted the function *iter*, so this order can easily be constructed. The order is used in the quantified state update together with a last-win clash-semantics to obtain the desired behavior.

6.2 Comparison to Traditional Dependence Analysis

Our dependence analysis is different from most existing analyses for loop parallelization in compilers [1, 28]. The major difference is that these analyses must not be expensive in terms of computation time, because the user waits for the compiler to finish. Traditionally, precision is traded off for lower cost. Here we use dependence information to avoid using induction which comes with an extremely high cost, because it typically requires user interaction. In consequence, we strive to make the dependence analysis as precise as possible as long as it is still fully automatic. In particular, our analysis can afford to try several algorithms that work well for different classes of loops.

A second difference to traditional dependence analysis is that we do not require a definite answer. When used during compilation to a parallel architecture, a dependence analysis must give a Boolean answer as to whether a given loop is parallelizable or not. In our setting it is useful to know that a loop is parallelizable relative to satisfaction of a symbolic constraint. Then we can let a theorem prover validate or refute this constraint, which typically is a much easier problem than proving the original loop.

7 Implementation of the Dependence Analysis

Our dependence analysis analyzes a loop and symbolically computes a *constraint* that characterizes when the loop is free of dependencies. The advantage of the constraint-based approach is that we can avoid to deal with a number of very hard problems such as aliasing: for example, locations `a[i]` and `b[i]` are the same iff `a` and `b` are references to the same array, which can be difficult to determine. Our analysis side-steps the aliasing problem simply by generating a constraint saying that *if* `a` is not the same array as `b` *then* there is no dependence. The validity of the generated constraint will then be decided by a theorem prover.

When looking for dependencies in the loop we do not analyze the loop itself but the state updates computed from the generic loop iteration. The dependence analysis is, therefore, defined over updates. The binary function δ defined in Table 1 takes two updates as arguments and computes a constraint that characterizes the absence of data flow-dependence among its arguments. In the definitions, we let *locs(t)* be the set of locations occurring in the term *t* and *slocs(`loc`)* the set of locations occurring as arguments in `loc` as defined below:

$$\begin{aligned} slocs(\mathtt{v}) &= \emptyset \\ slocs(\mathtt{o.f}) &= locs(o) \\ slocs(\mathtt{a[i]}) &= locs(a) \cup locs(i) \end{aligned}$$

The computation of the dependence constraint of a loop uses the vectors $\vec{\Gamma}$ and $\vec{\mathcal{U}}$ extracted from the success leaves during symbolic execution of the loop body. They were obtained as the result of a generic loop iteration in Section 4. If the preconditions of two leaves are true for two different loop iterations we need to ensure that the updates of the leaves are data flow-independent of each other (Def. 3). Formally, if there exist two distinct iteration numbers k and l and (possibly identical) leaves r and s, for which $k < l$, $\{\mathtt{i} := iter(k)\}\Gamma_r$ and $\{\mathtt{i} := iter(l)\}\Gamma_s$ are true, then we need to ensure independence of $\mathcal{U}_r$ and $\mathcal{U}_s$.

We do this for all pairs of leaves and define the dependence constraint for the entire loop as follows where GR is the loop range predicate and $I_{r,s,k,l}$ is defined as $\{\mathtt{i} := iter(k)\}\,\mathcal{U}_r\ \delta\ \{\mathtt{i} := iter(l)\}\,\mathcal{U}_s$.

$$\mathcal{C} \equiv \bigwedge_{r,s} \forall k,l.\ \left(k < l \wedge \left(\begin{array}{c} GR_k \wedge GR_l \wedge \\ \{\mathtt{i} := iter(k)\}\Gamma_r \wedge \{\mathtt{i} := iter(l)\}\Gamma_s \end{array}\right)\right) \rightarrow I_{r,s,k,l}$$

The condition $k < l$ ensures that we only capture data flow-dependence and not data anti-dependence.

Table 1. Computing dependence constraints among updates.

	Atomic updates
$\mathtt{v} := \mathtt{val}\ \delta\ \mathtt{loc} := \mathtt{val}'$	$= \begin{cases}\mathtt{true} & \textbf{when } \mathtt{v} \notin (locs(\mathtt{val}') \cup slocs(\mathtt{loc})) \\ \mathtt{false} & \textbf{otherwise}\end{cases}$
$\mathtt{o1.f} := \mathtt{val}\ \delta\ \mathtt{loc} := \mathtt{val}'$	$= \neg(\bigvee_{o2.f \in (locs(\mathtt{val}') \cup slocs(\mathtt{loc}))} \mathtt{o1} \doteq \mathtt{o2})$
$\mathtt{a[i]} := \mathtt{val}\ \delta\ \mathtt{loc} := \mathtt{val}'$	$= \neg(\bigvee_{b[j] \in (locs(\mathtt{val}') \cup slocs(\mathtt{loc}))} (\mathtt{a} \doteq \mathtt{b} \wedge \mathtt{i} \doteq \mathtt{j}))$
	General updates
$\mathcal{U}\ \delta\ \backslash\mathtt{if}\ (b)\ \{\mathcal{U}'\}$	$= \neg b \vee \mathcal{U}\ \delta\ \mathcal{U}'$
$\backslash\mathtt{if}\ (b)\ \{\mathcal{U}\}\ \delta\ \mathcal{U}'$	$= \neg b \vee \mathcal{U}\ \delta\ \mathcal{U}'$
$\mathcal{U}\ \delta\ \backslash\mathtt{for}\ T\ s;\ \mathcal{U}'(s)$	$= \forall s.\ \mathcal{U}\ \delta\ \mathcal{U}'(s)$
$\backslash\mathtt{for}\ T\ s;\ \mathcal{U}(s)\ \delta\ \mathcal{U}'$	$= \forall s.\ \mathcal{U}(s)\ \delta\ \mathcal{U}'$
$\mathcal{U}_0, \ldots, \mathcal{U}_m\ \delta\ \mathcal{U}'_0, \ldots, \mathcal{U}'_n$	$= \bigwedge_{i,j} \mathcal{U}_i\ \delta \mathcal{U}'_j$

Example 10. Consider the loop from the array reversal Example 1. When computing the effect of the generic loop iteration, we get one success leaf with the following update: $\{\mathtt{tmp} := \mathtt{a[i]}, \mathtt{a[i]} := \mathtt{a[a.length\ -\ 1\ -\ i]}, \mathtt{a[a.length\ -\ 1\ -\ i]} := \mathtt{a[i]}\}$.

The dependence constraint $I_{0,0,k,l}$ is false only if $\mathtt{a.length\ -\ 1\ -}\ iter(l) \doteq iter(k)$ holds. In the example we have $iter(n) = n$, so this can be simplified to $\mathtt{a.length\ -\ 1} \doteq k + l$.

In order for $\mathcal{C}$ to be true we need to show that there are no iteration numbers k and l, such that the above equality holds. From the guard specification we obtain that the maximum iteration number is `a.length / 2 - 1`. The maximum value of $k + l$ is, therefore, `a.length - 3` which is not equal to `a.length - 1`. This makes $\mathcal{C}$ true and means that the loop does not contain any dependencies that cannot be handled by our method. □

Computing $\boldsymbol{mod}(\vec{\mathcal{U}}, \mathcal{S})$

In Section 4 we used $mod(\vec{\mathcal{U}}, \mathcal{S})$ to compute the set of those locations in $\mathcal{S}$ whose assigned term in $\mathcal{U}$ differs from its assigned term in $\mathcal{S}$. This is very similar to an output dependence analysis. If a location is assigned a different term in $\mathcal{U}$ and $\mathcal{S}$ there will be an output dependence between them. Similarly as above, we define in Table 2 a function δ^o that gives the set of locations, where the terms in its two update arguments differ. The fourth case in the part for atomic updates in Table 2 is the default that is used when none of the other cases applies.

It is sometimes not possible to decide the **when** side conditions in Table 2. In this case we approximate conservatively and assume they are true. Possibly, we remove too much information this way, but the method remains sound. If the second argument is a quantified update, the set of locations could potentially be very large which would make the computation of δ^o very expensive. This can, however, not happen since quantified updates cannot occur in the updates computed for the generic loop iteration.

Table 2. Computing output dependence constraints.

	Atomic updates
$\mathtt{v}{:=}\mathtt{val}\ \delta^o\ \mathtt{v}{:=}\mathtt{val'}$	$= \{\mathtt{v}\}$ **when** $\mathtt{val} \not\doteq \mathtt{val'}$
$\mathtt{o.f}{:=}\mathtt{val}\ \delta^o\ \mathtt{o'.f}{:=}\mathtt{val'}$	$= \{\mathtt{o'.f}\}$ **when** $\mathtt{o} \doteq \mathtt{o'} \wedge \mathtt{val} \not\doteq \mathtt{val'}$
$\mathtt{a[i]}{:=}\mathtt{val}\ \delta^o\ \mathtt{b[j]}{:=}\mathtt{val'}$	$= \{\mathtt{b[j]}\}$ **when** $\mathtt{a} \doteq \mathtt{b} \wedge \mathtt{i} \doteq \mathtt{j} \wedge \mathtt{val} \not\doteq \mathtt{val'}$
$\_\ \delta^o\ \_$	$= \emptyset$
	General updates
$\mathcal{U}\ \delta^o\ \backslash\mathtt{if}\ (b)\ \{\mathcal{U}'\}$	$= \mathcal{U}\ \delta^o\ \mathcal{U}'$ **when** b
$\backslash\mathtt{if}\ (b)\ \{\mathcal{U}\}\ \delta^o\ \mathcal{U}'$	$= \mathcal{U}\ \delta^o\ \mathcal{U}'$ **when** b
$\mathcal{U}\ \delta^o\ \backslash\mathtt{for}\ T\ s;\ \mathcal{U}'(s)$	$= \bigcup_s \mathcal{U}\ \delta^o\ \mathcal{U}'(s)$
$\backslash\mathtt{for}\ T\ s;\ \mathcal{U}(s)\ \delta^o\ \mathcal{U}'$	$= \bigcup_s \mathcal{U}(s)\ \delta^o\ \mathcal{U}'$
$\mathcal{U}_0, \ldots, \mathcal{U}_m\ \delta^o\ \mathcal{U}'_0, \ldots, \mathcal{U}'_n$	$= \bigcup_{i,j} \mathcal{U}_i\ \delta^o\ \mathcal{U}'_j$

Another possibility when a side condition cannot be decided would be to compute two different results, one result for when the condition is true and one for when it is false. A problem with this approach is that it potentially doubles the number of returned results each time a side condition cannot be decided. The returned result is used for the computation of the generic loop iteration and, therefore, returning many results would lead to many different generic loop iterations where each needs to be analyzed by the dependence analysis.

Further details on the implementation of the dependence analysis are in [25].

8 Constructing the State Update

If we can show that the iterations of a loop are independent of each other (i.e., the constraint $\mathcal{C}$ defined in the previous section holds), we can capture all state modifications of the loop in one update. Concretely, we use the following quantified update (GR_n, Γ_r, *iter*, and $\mathcal{U}_r$ were defined in Sections 4 and 5):

$$\mathcal{U}_{loop} \equiv \backslash\mathtt{for}\ \mathbf{int}\ n;\ \{\backslash\mathtt{if}\ (GR_n)\ \{\{\mathtt{i}{:=}iter(n)\}\bigcup_r \backslash\mathtt{if}\ (\Gamma_r)\ \{\mathcal{U}_r\}\}\} \tag{4}$$

The innermost conditional update in (4) corresponds to one loop iteration, where the loop variable $\mathtt{i}$ has the value $iter(n)$. In each state only one Γ_r can be true so we do not need to ensure any particular order of the updates $\vec{\mathcal{U}}$.

The guard GR ensures that the iteration number n is within the loop range. We must take care when using last-win clash-semantics to handle data output-dependence. The iteration with the highest iteration number should have priority over all other iterations. This is ensured by the standard well-order on the JAVA integer type.

9 Using the Analysis in a Correctness Proof

When we encounter a loop during symbolic execution we analyze it for parallelizability as described above and compute the dependence constraint. We replace the loop by (4) if no failed leaves for the iteration statement or the guard expression can be reached (see Section 4), the loop terminates (formula GT, see Section 5), and the dependence constraint $\mathcal{C}$ in Section 7 is valid. Taken together, this yields:

$$\begin{aligned}\mathcal{D} \equiv (\bigwedge_{\mathcal{F}\in\vec{\mathcal{F}}} &\neg(\exists n.GR_n \wedge \{i{:=}iter(n)\}\mathcal{F})) \wedge \\ &\neg(\exists n.GR_n \wedge \{i{:=}iter(n)\}GF) \wedge GT \wedge \mathcal{C}\end{aligned}$$

If $\mathcal{D}$ does not hold, we fall back to the standard rules to verify the loop (usually induction). In many cases it is not trivial to immediately validate or refute $\mathcal{D}$. Then we perform a cut on $\mathcal{D}$ in the proof and replace the loop by the quantified state update $\mathcal{U}_{loop}$ (4) in the proof branch where $\mathcal{D}$ is assumed to hold. The general outline of a proof using a cut on $\mathcal{D}$ is as follows:

$$\cfrac{\cfrac{\text{If not } \Gamma \Rightarrow \mathcal{D}, \text{ use standard induction}}{\Gamma \Rightarrow \mathcal{U}\langle \textbf{for } \ldots ; \ldots\rangle\phi, \mathcal{D}} \qquad \cfrac{\Gamma, \mathcal{D} \Rightarrow \mathcal{U}\mathcal{U}_{loop}\langle\ldots\rangle\phi}{\Gamma, \mathcal{D} \Rightarrow \mathcal{U}\langle \textbf{for } \ldots ; \ldots\rangle\phi}}{\cfrac{\Gamma \Rightarrow \mathcal{U}\langle \textbf{for } \ldots ; \ldots\rangle\phi}{\vdots}}\; cut$$

If we can validate or refute $\mathcal{D}$ we can close one of the two branches. Typically, this involves to show that there is no aliasing between the variables occurring in the dependence constraint. Even when it is not possible to prove or to refute $\mathcal{D}$ our analysis is useful, because $\mathcal{D}$ in the succedent of the left branch can make it easier to close.

10 Evaluation

We evaluated our method with three representative JAVA CARD programs [19]: DeMoney, SafeApplet and IButtonAPI that together consist of ca. 2200 lines of code (not counting comments). These programs contain 17 loops. Out of these, our method can be applied to five (sometimes, a simple code transformation like `v += e` to `v = v0 + i * e` is required). Additionally, four loops can be handled if we allow object creation in the quantified updates (which is currently not realized). The remaining eight loops cannot be handled because they contain abrupt termination and irregular step functions. The results are summarized in Table 3.

All loops in the row "handled" are detected automatically as parallelizable and are transformed into quantified updates. The evaluation shows that a considerable number of loops in realistic legacy programs can be formally verified without resorting to interactive and, therefore, expensive techniques such as induction. Interestingly, the percentage of loops that can be handled differs drastically among the three programs. A closer

Table 3. Parallelizable loops in some representative JAVA CARD programs.

	DeMoney	SafeApplet	IButtonAPI	Total
LoC	1633	514	102	2249
Size (kB)	182	22	3	207
# loops	10	6	1	17
handled	4	0	1	5
with ext.	3	1	0	4
remaining	3	5	0	8

inspection reveals that the reason is not that, for example, all the loops in SafeApplet are inherently not parallelizable. Some of them could be rewritten so that they become parallelizable. This suggests to develop programming guidelines (just as they exist for compilation on parallel architectures) that ensure parallelizability of loops.

11 Future Work

The coverage of our verification method can be improved in various ways. One example is the function from the iteration number to the value of the loop variable (see Section 5). In addition, straightforward automatic program transformations that reduce the amount of dependencies (for example, `v += e;` into `v = vInit + i * e;`) could be derived by looking at the updates computed from a generic loop iteration. Recent work on automatic termination analysis [11] could be tried in the present setting for proving the termination constraint in Section 5.

We intend to develop general programming guidelines that ensure parallelizability of loops. The current trend towards multi-core processors will result in more code being written in such a way that it is parallelizable and will for sure rekindle the interest in parallelizability.

Critical dependencies exhibited during dependence analysis are likely to cause complications even in a proof attempt based on a more general proof method such as invariants or induction. Hence, one could try to use the information obtained from the dependence analysis to guide the generalization of, for example, loop invariants.

At the moment we take into account JAVA integer semantics only by checking for overflow. The integer model could be made more precise by computing all integer operators modulo the size of the underlying integer type. This would require changes only in the dependence analysis; the JAVA DL calculus covers full JAVA integer semantics already [5].

So far our verification method has been worked out and implemented for loop structures, however, it can be seen as a particular instance of a modular approach to proving correctness of non-linear programs composed of code pieces `p(i)` parameterized by some `i`:

1. Compute automatically the effect $\mathcal{U}_{\mathtt{p}}(\mathtt{i})$ of `p(i)` with respect to a given precondition.

2. Using the dependence analysis on $\mathcal{U}_{\mathtt{p}}(\mathtt{i})$, compute a sufficient condition $\mathcal{C}$ under which the code `p(i)` can be seen as *modular* with respect to different iterations of the parameter `i`.
3. The result of the analysis can be used in non-linear composition of `p(i)` as done here for iterative control structures. The idea is just as well applicable for recursive method calls and concurrent processes as is illustrated by the following example:

   ```
   int[] a = new int[n];
   for (int i = 0; i < n; i++) {
     new MyThread(i,a).start();
   }
   ```

   If we assume that the `run()` method of the class `MyThread` updates exactly position `i` of the array `a`, then the effect can be easily captured by an update obtained from executing `run()` in the instance created by **new** `MyThread(i,a);`. One difference to loops is that in the context of threads one would probably exclude data output-dependence (see Section 6.1) unless assumptions about the scheduler can be made. Otherwise, the inherently parallel structure of state updates is well suited to model concurrent threads.

In this paper we do not discuss in detail what happens after a loop has been transformed into a quantified update. This is outside the scope of the present work. So far, the KeY theorem prover has limited capabilities for automatic reasoning over first-order quantified updates. Since quantified updates occur in many other scenarios [24] it is worth to spend more effort on that front.

12 Conclusion

We presented a method for formal verification of loops that works by transforming loops into automatable first-order constructs (quantified updates) instead of interactive methods such as invariants or induction. The approach is restricted to loops that can be parallelized, but an analysis of representative programs from the JAVA CARD domain shows that such loops occur frequently. The method can be applied to most initialization and array copy loops but also to more complex loops as witnessed by Example 1.

The method relies on the capability to represent state change information effecting from symbolic execution of imperative programs explicitly in the form of syntactic updates [3, 24]. With the help of updates the effect of a generic loop iteration is represented so that it can be analyzed for the presence of data dependencies. Ideas for the dependency analysis are taken from compiler optimization for parallel architectures, but the analysis is not merely static. Loops that are found to be parallelizable are transformed into first-order quantified updates to be passed on to an automated theorem prover.

A main advantage of our method is its robustness in the presence of syntactic variability in the target programs. This is achieved by performing symbolic execution before doing the dependence analysis. The method is also fully automatic whenever it is applicable and gives useful results in the form of symbolic constraints even if it fails.

Acknowledgments

Many thanks to Richard Bubel whose help with the prototypic implementation was invaluable. Max Schröder did the final implementation [25] which is the basis for Section 7. Thanks are also due to Philipp Rümmer for many inspiring discussions. The valuable comments of the anonymous reviewer led to several improvements.

References

1. U. Banerjee, S.-C. Chen, D. J. Kuck, and R. A. Towle. Time and parallel processor bounds for Fortran-like loops. *IEEE Trans. Computers*, 28(9):660–670, 1979.
2. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: an overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Post Conference Proceedings of CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices, Marseille*, volume 3362 of *LNCS*, pages 49–69. Springer, 2005.
3. B. Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, Cannes, France*, LNCS 2041, pages 6–24. Springer, 2001.
4. B. Beckert, R. Hähnle, and P. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2007.
5. B. Beckert and S. Schlager. Software verification with integrated data type refinement for integer arithmetic. In E. A. Boiten, J. Derrick, and G. Smith, editors, *Proc. , Intl. Conf. on Integrated Formal Methods*, volume 2999 of *LNCS*, pages 207–226. Springer, 2004.
6. B. Beckert, S. Schlager, and P. H. Schmitt. An improved rule for while loops in deductive program verification. In K.-K. Lau, editor, *Proc. , Seventh Intl. Conf. on Formal Engineering Methods (ICFEM), Manchester, UK*, LNCS. Springer-Verlag, 2005.
7. R. S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press, 1988.
8. C.-B. Breunesse. *On JML: Topics in Tool-assisted Verification of Java Programs*. PhD thesis, Radboud University of Nijmegen, 2006.
9. A. Bundy, D. Basin, D. Hutter, and A. Ireland. *Rippling: Meta-Level Guidance for Mathematical Reasoning*, volume 56 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, June 2005.
10. L. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness: a developer-oriented approach. In *Proc. Formal Methods Europe, Pisa, Italy*, volume 2805 of *LNCS*, pages 422–439. Springer, 2003.
11. B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In M. I. Schwartzbach and T. Ball, editors, *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation, Ottawa, Canada*, pages 415–426. ACM Press, 2006.
12. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proc. ACM SIGPLAN 2002 Conf. on Programming Language Design and Implementation, Berlin*, pages 234–245. ACM Press, 2002.
13. T. Gedell and R. Hähnle. Automating verification of loops by parallelization. In M. Hermann and A. Voronkov, editors, *Proc. Intl. Conf. on Logic for Programming Artificial Intelligence and Reasoning, Phnom Penh, Cambodia*, volume 4246 of *LNCS*, pages 332–346, Oct. 2006.
14. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.
15. G. J. Holzmann. Software analysis and model checking. In E. Brinksma and K. G. Larsen, editors, *Proc. Intl. Conf. on Computer-Aided Verification CAV, Copenhagen*, volume 2402 of *LNCS*, pages 1–16. Springer, July 2002.

16. B. Jacobs, C. Marché, and N. Rauch. Formal verification of a commercial smart card applet with multiple tools. In C. Rattray, S. Maharaj, and C. Shankland, editors, *Proc. 10th Conf. on Algebraic Methodology and Software Technology (AMAST), Stirling, UK*, volume 3116 of *LNCS*, pages 241–257. Springer, July 2004.
17. K. R. M. Leino and F. Logozzo. Loop invariants on demand. In K. Yi, editor, *Proc. Progr. Lang. and Systems, 3rd Asian Symposium, Tsukuba*, volume 3780 of *Lecture Notes in Computer Science*, pages 119–134. Springer-Verlag, 2005.
18. C. Marché and C. Paulin-Mohring. Reasoning about Java programs with aliasing and frame conditions. In J. Hurd and T. Melham, editors, *Proc. 18th Intl. Conference on Theorem Proving in Higher Order Logics, Oxford, UK*, volume 3603 of *LNCS*, pages 179–194. Springer, Aug. 2005.
19. W. Mostowski. Formalisation and verification of Java Card security properties in dynamic logic. In M. Cerioli, editor, *Proc. Fundamental Approaches to Software Engineering (FASE), Edinburgh*, volume 3442 of *LNCS*, pages 357–371. Springer, Apr. 2005.
20. O. Olsson and A. Wallenburg. Customised induction rules for proving correctness of imperative programs. In B. Beckert and B. Aichernig, editors, *Proc. , Software Engineering and Formal Methods (SEFM), Koblenz, Germany*, pages 180–189. IEEE Press, 2005.
21. A. Platzer. Using a program verification calculus for constructing specifications from implementations. Master's thesis, Univ. Karlsruhe, Dept. of Computer Science, 2004.
22. A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *Programming Languages and Systems (ESOP)*, volume 1576 of *LNCS*, pages 162–176. Springer, 1999.
23. E. Rodríguez-Carbonell and D. Kapur. Program verification using automatic generation of invariants. In Z. Liu and K. Araki, editors, *Proc. Theoretical Aspects of Computing (ICTAC), Guiyang, China, Revised Selected Papers*, volume 3407 of *LNCS*, pages 325–340. Springer-Verlag, 2005.
24. P. Rümmer. Sequential, parallel, and quantified updates of first-order structures. In M. Hermann and A. Voronkov, editors, *Proc. Logic for Programming, Artificial Intelligence and Reasoning, Phnom Penh, Cambodia*, volume 4246 of *LNCS*, pages 422–436. Springer, 2006.
25. M. Schroeder. Using a symbolic dependence analysis for verification of programs containing loops. Master's thesis, Department of Computer Science, University of Karlsruhe, 2007.
26. K. Stenzel. *Verification of Java Card Programs*. PhD thesis, Fakultät für angewandte Informatik, University of Augsburg, 2005.
27. Sun Microsystems, Inc., Santa Clara, California, USA. JAVA CARD *2.2.1 Application Programming Interface*, Oct. 2003.
28. M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. The MIT Press, 1989.