Automating Verification of Loops by Parallelization*

Tobias Gedell and Reiner Hähnle

Department of Computing Science, Chalmers University of Technology
S-412 96 Göteborg, Sweden, {gedell,reiner}@chalmers.se

Abstract. Loops are a major bottleneck in formal software verification, because they generally require user interaction: typically, induction hypotheses or invariants must be found or modified by hand. This 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, where a loop can be parallelized whenever it avoids dependence of the loop iterations from each other. We develop a dependence analysis that ensures parallelizability. It guarantees soundness of a proof rule that transforms a loop into a universally quantified update of the state change information represented by the loop body. This makes it possible to use automatic first order reasoning techniques to deal with loops. The method has been implemented in the KeY verification tool. We evaluated it with representative case studies from the JAVA CARD domain.

1 Introduction

It is generally agreed that loops and recursive calls are the main bottleneck in formal software verification. The source of the problem is that loops and recursion are proof theoretically handled either with invariant rules or with induction. In both cases, it is necessary in general to strengthen invariants and induction hypotheses in order to make proofs go through. There are also many technicalities with those rules that make their application difficult. A number of heuristic techniques have been developed to guide induction proofs and to find appropriate induction hypotheses (for example, [6, 8]).

The context of the present work is formal verification of functional properties of sequential JAVA programs [1]. Here the situation is aggravated by the fact that the above mentioned techniques have been developed for relatively simple functional programming languages and are not readily applicable to a complex, imperative, object-based language such as JAVA (similar comments apply to C, C++, or C#). Hence, not only is there a lack of heuristic techniques that help to automate proofs about loops in JAVA, but due to the complexity of loop rules in imperative languages [5] user interaction involves a high amount of technical knowledge and is extremely expensive.

A recent divide-and-conquer technique for decomposition of induction proofs [15] works for imperative programs, but it is targeted at simplifying user interaction rather

* 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.

than eliminating it. In order to deal *automatically* with loops in verification of JAVA-like languages there are not too many options at present: abstraction [13] and approximation [10] are incomplete and in some scenarios even unsound. They also impose limits on what can be expressed in specifications. If the number of loop iterations is known and small then it is possible to use symbolic execution with finite unwinding [11]. The state of the art in JAVA verification is, however, that complex user interaction is unavoidable for almost all loops [7].

In this paper we present an *automatic* deductive verification technique that is applicable to many loops occurring in practically relevant JAVA programs. Like any automatic method it cannot handle all loops, but it is seamlessly integrated with a complete interactive verification system. In addition, it computes useful information even when it fails. To make things concrete, we look at an example (where `e(i)` is an expression with an occurrence of `i`): **for** `(`**int** `i = 0; i < a.length; i++)a[i] = e(i);`

The effect of this piece of code is simply to initialize all elements of the array `a` with the expression $e(\texttt{i})$ at index `i`. Since the length of `a` is in general unknown, it is not possible to deal with this loop by finite unwinding. An abstraction of this program has difficulties to record that the value `a.length` depends on `a`. On the other hand, in most cases it is overkill to use induction on such a simple problem. In order to describe the effect of such loops it is usually sufficient to be able to quantify universally over state update expressions that are performed in parallel. From a proof theoretic point of view, quantified state modifiers can be handled by skolemization and simplification [17], hence, they are amenable to automated proof search.

In general, the initialization, guard and step expressions, as well as the loop body could be more complicated than in the example above. We are looking for a technique that does not rely on the target program being in a particular syntactic form. Of course, we need to make certain assumptions to ensure that the effect of a loop is expressible as a quantified update. This problem is closely related to loop vectorization and parallelization and it is possible to use notions developed in these fields. The main issue is to exclude certain data dependencies. For example, in the case of $e(\texttt{i}) \equiv \texttt{a[i - 1]}$ the code above cannot be transformed into a quantified state update, because there exist dependencies between the updates.

The contribution of this paper is a deductive verification method for treating loops[1] based on the ideas just sketched. Its main properties are:

Robustness The target program needs not to be in a particular syntactic form. This is achieved by computing the accumulated effect of the expressions and statements occurring in the loop by symbolic execution *before* checking the dependencies in the loop body (Section 4 and Section 5).

Soundness There is an automatic dependence analysis that guarantees sound applicability (Section 6).

Automation Proof theoretic treatment of the effect of loops is not by induction but by universally quantified state modification and is automatic (Section 7).

[1] The technique is applicable both to for- and while-loops. In this presentation we concentrate on the former to make the presentation more concise, and because for-loops are much more common in our application domain JAVA CARD.

Relevance The method applies not only to a few academic examples, but to a substantial number of loops in realistic programs. An experimental evaluation of a number of realistic JAVA CARD programs confirms this (Section 9).

2 Basic Definitions

The platform for our experiments is the KeY tool [1], 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) [12] 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 axiomatizes full JAVA minus multi-threading, floating point types, and dynamic class loading.

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 in JAVA DL formulas are basically 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 `o1.f` $\doteq$ `1`, where $\doteq$ denotes equality, 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. 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* $\mathtt{loc} := \mathtt{val}$*, 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* $\{\mathtt{loc} := \mathtt{val}\}\phi$ *is the same as that of* $\langle \mathtt{loc=val;} \rangle \phi$*.*

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) $\mathcal{U}\mathcal{U}'$ (*applied on update*), (iv)* $\backslash\texttt{if}\ (b)\ \{\mathcal{U}\}$*, where b is a quantifier-free formula (*conditional execution*), (v)* $\backslash\texttt{for}\ T\ s; \mathcal{U}(s)$*, where s is a variable over a well-ordered type T and $\mathcal{U}(s)$ is an update with occurrences of s (*quantification*).*

The semantics of sequential updates, conditional updates and updates applied on 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 $\backslash\texttt{for}\ T\ s; \mathcal{U}(s)$ is the parallel execution of all updates in $\bigcup_{x\in T}\{\mathtt{s}:=x; \mathcal{U}(s)\}$. As for parallel updates, a last-win clash-semantics is in place: the maximal 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 limiting: 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 [17].

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 $\mathcal{S} = \{\mathtt{i}:=1\}$. The remaining loop now has the form: **for** `(; i < a.length; i++)...`

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 $\mathcal{S}' = \{\mathtt{i}:=\mathtt{i}+1, \backslash\texttt{if}\ (c \neq 0)\ \{\mathtt{a[i]}:=\mathtt{b[i+1]}\}, \backslash\texttt{if}\ (c \doteq 0)\ \{\mathtt{a[i]}:=\mathtt{b[i-1]}\}\}$, 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. In the example it results in a logical constraint $\mathcal{C}$ that, among other things, contains the disjunction of $\texttt{c} \not\doteq 0$ and $\texttt{a} \not\doteq \texttt{b}$. 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, $\texttt{b.length} > \texttt{a.length}$.

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 $\{\texttt{c}:=1\}$ and the termination constraint in $\mathcal{D}$ holds. In this case, we can additionally simplify $\mathcal{S}'$ to $\{\texttt{c}:=1,\ \texttt{i}:=\texttt{i}+1,\ \texttt{a[i]}:=\texttt{b[i+1]}\}$.

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 7. The result for the example is:

$$\texttt{\textbackslash for}\ \textbf{int}\ I;\ \{\texttt{i}:=I\}\{\texttt{\textbackslash if}\ (\texttt{i} \geq 1 \wedge \texttt{i} < \texttt{a.length})\ \{\texttt{c}:=1,\ \texttt{i}:=\texttt{i}+1,\ \texttt{a[i]}:=\texttt{b[i+1]}\}\}$$

The `for`-expression is a universal first order quantifier whose scope is an update that contains occurrences of the variable `i` (see Def. 2 and [17]). 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 $j < \texttt{a.length}$ we obtain immediately the update $\texttt{a[}j\texttt{]}:=\texttt{b[}j\texttt{+1]}$ by instantiation. Proof search is performed by the usual first order strategies without user interaction.

4 Computing state modifications

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 [16] 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.

Let $\mathtt{p}$ be the code in (2). The main idea is to try to prove validity of the program formula $\mathcal{S}\langle\mathtt{p}\rangle F$, where F is an arbitrary but unspecified non-rigid predicate that signifies when to stop symbolic execution. Symbolic execution of $\mathtt{p}$ starting in state $\mathcal{S}$ eventually yields a proof tree whose open leaves are of the form $\Gamma \to \mathcal{U}F$ for some update expression $\mathcal{U}$. The predicate F cannot be shown to be true or false in the program logic. Therefore, after all instructions in $\mathtt{p}$ 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 computation branches in the loop body.

Example 1. Consider the following statement $\mathtt{p}$:

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

After the attempt to prove $\langle\mathtt{p}\rangle F$ becomes stuck, i.e. all instructions have been symbolically executed, there are two open leaves:

$$V \wedge i > 2 \to \{\mathtt{a[i]} := 0, \mathtt{i} := \mathtt{i} + 1\}F$$
$$V \wedge i \not> 2 \to \{\mathtt{a[i]} := 1, \mathtt{i} := \mathtt{i} + 1\}F$$

where V stands for $\neg(\mathtt{a} = \textbf{null}) \wedge i \geq 0 \wedge \mathtt{i} < \mathtt{a.length}$. From these we extract the following vectors:

$$\vec{\Gamma} \equiv \langle V \wedge i > 2, V \wedge i \not> 2\rangle$$
$$\vec{\mathcal{U}} \equiv \langle\{\mathtt{a[i]} := 0, \mathtt{i} := \mathtt{i} + 1\}, \{\mathtt{a[i]} := 1, \mathtt{i} := \mathtt{i} + 1\}\rangle$$

□

If the loop iteration throws an exception, abruptly terminates the loop, or when the automatic strategies are not strong enough to execute all instructions in $\mathtt{p}$ to completion, some open leaf will contain unhandled instructions and be of a form different from $\Gamma \to \mathcal{U}F$. We call these *failed leaves* in contrast to leaves of the form $\Gamma \to \mathcal{U}F$ that are called *successful*.

If a failed leaf can be reached from the initial state, 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. In order to do this we create a vector $\vec{\mathcal{F}}$ consisting of the Γ extracted from all failed leaves and let the negation of $\vec{\mathcal{F}}$ become a condition that needs to be proven when applying our method.

Example 2. In Example 1 only the successful leaves are shown. When all instructions have been symbolically executed, there are in addition failed leaves of following form:

$$\begin{array}{ll} \mathtt{a} \doteq \textbf{null} & \to \ldots F \\ \mathtt{a} \not\doteq \textbf{null} \wedge \mathtt{i} < 0 & \to \ldots F \\ \mathtt{a} \not\doteq \textbf{null} \wedge \mathtt{i} \not< \mathtt{a.length} & \to \ldots F \end{array}$$

From these we extract the following vector:

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

□

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 3. 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 $\mathcal{S} = \{\texttt{c}:=1, \texttt{i}:=0\}$. 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 $\mathcal{S}$ 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 get $\{\backslash\textbf{if } (\texttt{c} \neq 0)\ \{\texttt{a[i]}:=0\}, \texttt{b[i]}:=0, \texttt{i}:=\texttt{i}+1\}$, 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 at least one of the branches we backtrack to state $\mathcal{S}_{\text{init}}$ and use the standard rules to prove the loop.
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 successful leaf, more formally: $\mathcal{S}' = \{(\ell:=e) \in \mathcal{S} \mid \ell \notin \mathit{mod}(\vec{\mathcal{U}})\}$, where $\mathit{mod}(\vec{\mathcal{U}})$ is the set of locations whose value in $\vec{\mathcal{U}}$ differs from its value in $\mathcal{S}$.
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 and all methods called by the loop. 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 4. Example 3 yields the following sequence of states:

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

□

Computing the set $mod(\vec{\mathcal{U}})$ 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. One can use a simplified version of the dependence analysis described in Section 6 (modified to yield always a boolean answer) to obtain an approximation of location collision. The dependence analysis always terminates so this does not change the overall termination behavior.

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. Here we make a safe approximation and remove `a.f` unless we can show that `a` and `b` refer to different objects.

5 Loop Variable and Loop Range

For the dependence analysis and for creating the quantified state update later we need to identify the loop variable and the loop range. In addition, we need to know the value that the loop variable has in each iteration of the loop, that is, the function from the iteration number to the value of the loop variable in that iteration. This is a hard problem in general, but whenever the loop variable is incremented or decremented with a constant value in each iteration, it is easy to construct this function. At present we impose this as a restriction: the update of the loop variable must have the form `l := l op e`, where `l` is the loop variable and `e` is invariant during loop execution. It would be possible to let the user provide this function at the price of making the method less automatic.

To identify the loop variable we compute a set of candidate pairs $(\texttt{l}, \texttt{e})$ where `l` is a location that is assigned the expression `e`, satisfying the above restriction, in all successful leaf nodes of the generic iteration. Formally, this set is defined as $\{(\texttt{l}, \texttt{e}) \mid \bigwedge_{\mathcal{U}\in\vec{\mathcal{U}}}\{\texttt{l}:=\texttt{e}\} \in \mathcal{U}\}$. The loop variable is supposed to have an effect on the loop range; therefore, we remove all those locations from the candidate set that do not occur in the guard. If the resulting set consists of more than one location, we arbitrarily choose one.

The remaining candidates should be eliminated, because they will all cause data flow-dependence. 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 + I * c;`, where `init` is the initial value of `l` and `I` the iteration number.

Example 5. Consider the code in Example 1 which gives the following vector $\vec{\mathcal{U}}$ of updates occurring in successful leaves:

$$\vec{\mathcal{U}} \equiv \langle \{\texttt{a[i]}:=0, \texttt{i}:=\texttt{i}+1\}, \{\texttt{a[i]}:=1, \texttt{i}:=\texttt{i}+1\} \rangle$$

We identify the location `i` as the loop variable, assuming that `i` occurs in the guard. □

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 $\langle$**boolean** `dummy = guard;`$\rangle\, F$. From the open leaves of the form $\Gamma \rightarrow \{\texttt{dummy} := \texttt{e}, \ldots\} F$, we create the formula *GS* which characterizes when the guard is true. Formally, *GS* is defined as $\bigvee_{\Gamma \in \vec{\Gamma}} (\Gamma \wedge \texttt{e} \doteq \textbf{true})$. 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 6. Consider the following guard $\texttt{g} \equiv \texttt{i < a.length}$. When all instructions in the formula $\langle$**boolean** `dummy = g;`$\rangle\, F$ have been symbolically executed, there are two successful leaves:

$$\texttt{a} \neq \textbf{null} \wedge \texttt{i} < \texttt{a.length} \rightarrow \{\texttt{dummy} := \textbf{true}\} F$$
$$\texttt{a} \neq \textbf{null} \wedge \texttt{i} \not< \texttt{a.length} \rightarrow \{\texttt{dummy} := \textbf{false}\} F$$

From these we extract the following formula *GS* (before simplification):

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

When the instructions have been executed, there is also the failed leaf $\texttt{a} \doteq \textbf{null} \rightarrow \ldots\ F$. From it we extract the following formula $GF \equiv \texttt{a} \doteq \textbf{null}$. □

After having computed the specification of the guard and identified the loop variable we determine the initial value *start* of the loop variable from the initial state $\mathcal{S}_{\text{init}}$. If an initial value cannot be found we let it be unknown. We try to determine the final value *end* of the loop variable from the successful leaves of the guard specification. Currently, we restrict this to guards of the form `l` op `e`. If we cannot determine the final value, we let it be unknown. We had already computed the *step* value during loop variable identification.

The formula *LR* characterizes when the value of `i` is within the loop range. It is defined as follows, which expresses that there exists an iteration with the particular value of the loop variable and that the iteration can be reached:

$$LR \equiv GS \wedge \exists n.\ \begin{pmatrix} n \geq 0 \wedge \texttt{i} \doteq start + n * step \wedge \\ \forall m.\ 0 \leq m < n \rightarrow \{\texttt{i} := start + m * step\} GS \end{pmatrix}$$

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

$$LT \equiv \exists n.\ n \geq 0 \wedge \{\texttt{i} := start + n * step\} \neg GS$$

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 [2, 18].

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 7. 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 update) that is read (appears on the right hand side 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 8. 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 7 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 7) 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. As we know the step expression of the loop variable, 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 [2, 18]. 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 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.

6.3 Implementation

Our dependence analysis consists of two parts. The first part analyzes the 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 second part of the dependence analysis is a tailor-made theorem prover that simplifies the integer equations occurring in the resulting constraints as much as possible.

The computation of the dependence constraints uses the vectors $\vec{\Gamma}$ and $\vec{\mathcal{U}}$ that represent successful leaves in the symbolic execution of the loop body and were obtained as the result of a generic loop iteration in Section 4. Let Γ_k and $\mathcal{U}_k$ be the precondition, respectively, the resulting update in the k-th leaf. If the preconditions of two leaves are true for different values in the loop range we need to ensure that the updates of the leaves are independent of each other (Def. 3). Formally, if there exist two distinct values K and L in the loop range and (possibly identical) leaves r and s, for which $\{\mathtt{i}:=K\}\Gamma_r$ and $\{\mathtt{i}:=L\}\Gamma_s$ are true, then we need to ensure independence of $\mathcal{U}_r$ and $\mathcal{U}_s$. We run our dependence analysis on $\mathcal{U}_r$ and $\mathcal{U}_s$ to compute the dependence constraint $\mathcal{C}_{r,s}$.

We do this for all pairs of leaves and define the dependence constraint for the entire loop as follows where LR is the loop range predicate:

$$\mathcal{C} \equiv \bigwedge_{r,s}((\exists K,L.\ (K \neq L \wedge \{\mathtt{i}:=K\}(LR \wedge \Gamma_r) \wedge \{\mathtt{i}:=L\}(LR \wedge \Gamma_s))) \rightarrow \mathcal{C}_{r,s})$$

Example 9. Consider the following loop that reverses the elements of the 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; }
```

When running the dependence analysis we get the following constraint:

$$\mathcal{C}_{0,0} \equiv \mathtt{a.length} < 2 \vee \mathtt{half} * 2 < \mathtt{a.length}$$

For this loop, the state $\mathcal{S}_{iter}$ contains $\mathtt{half}:=\mathtt{a.length\ /\ 2\ -\ 1}$ and the constraint is, therefore, simplified to $\mathtt{a.length} < 2 \vee (\mathtt{a.length}/2) * 2 < \mathtt{a.length} + 2$. This is simplified to **true** which makes $\mathcal{C}$ true and means that the loop does not contain any dependencies that cannot be handled by our method. □

7 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 (Def. 2). Concretely, we use the following quantified update (T is the type of the loop variable `i`; LR, Γ_r, $\mathcal{U}_r$ were defined in Sections 4 and 5):

$$\mathcal{U}_{loop} \equiv \backslash\mathtt{for}\ T\ I; \{\mathtt{i}:=I\}\{\backslash\mathtt{if}\ (LR)\ \{\bigcup_r \backslash\mathtt{if}\ (\Gamma_r)\ \{\mathcal{U}_r\}\}\} \tag{3}$$

The conditional update inside (3) corresponds to one loop iteration, where `i` has the value I. In each state only one Γ can be true so we do not need to ensure any particular order of the updates $\vec{\mathcal{U}}$.

The guard LR ensures that `i` is within the loop range. We must take care when using last-win clash-semantics to handle data output-dependence. When the step is positive, the iteration with the highest value of the loop variable should have priority over all other iterations. This is ensured by the standard well-order on the JAVA integer types.

A complication arises when the step is negative. Then we need to reverse the order so that the iteration with the lowest value of the loop variable has priority. Since each type has a fixed order we need to change the state update instead: it is sufficient to replace in (3) the update $\mathtt{i}:=I$ with $\mathtt{i}:=-I$.

8 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 (3)

if no failed leaves for the iteration statement or the guard expression can be reached (see Section 4), the loop terminates (formula LT, see Section 5), and the dependence constraint $\mathcal{C}$ in Section 6.3 is valid. Taken together, this yields:

$$\mathcal{D} \equiv \neg(\exists I.\ \{\mathtt{i} := I\}(LR \wedge \bigvee_i \mathcal{F}_i)) \wedge \neg GF \wedge LT \wedge \mathcal{C}$$

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 $\mathcal{U}_{loop}$ (3) 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:

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

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 succedent of the left branch can make it easier to close.

9 Evaluation

We evaluated our method with three representative JAVA CARD programs [14]: DeMoney, SafeApplet and IButtonAPI that together consist of ca. 2200 lines of code (not counting comments). In these programs there exist 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 the following table:

	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

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 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.

10 Conclusion

We presented a method for formal verification of loops that works by transforming loops into automizable 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 shown by Example 9.

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, 17]. 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.

Future Work The analysis can be improved in various ways. One example is the function from iteration number to 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 from a generic loop iteration. We also intend to develop general programming guidelines that ensure parallelizability of loops. Recent work on automatic termination analysis [9] could be adapted to the present setting for proving the termination constraint in Section 5.

Critical dependencies exhibited during the analysis are likely to cause problems as well in a proof attempt based on invariants or induction, so one could try to use the obtained information on dependencies to guide the generalization of loop invariants.

At the moment we observe JAVA integer semantics only by checking for overflow. The integer model could be made more precise by computing all integer operators modulo the the size of the underlying integer type. This would require changes in the dependence analysis; the JAVA DL calculus covers full JAVA integer semantic already [4].

Finally, the discussion in this paper stops after a loop has been transformed into a quantified update. So far, our theorem prover has limited capabilities for automatic reasoning over first order quantified updates. Since quantified updates occur in many other scenarios it is worth to spend more effort on that front.

Acknowledgments Many thanks to Richard Bubel whose help with the implementation was invaluable! Thanks are also due to Philipp Rümmer for many inspiring discussions.

References

1. W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool: integrating object oriented design and formal verification. *Software and System Modeling*, 4(1):32–54, 2005.
2. 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.
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 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.
5. 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.
6. R. S. Boyer and J. S. Moore. *A Computational Logic Handbook*. Academic Press, 1988.
7. C.-B. Breunesse. *On JML: Topics in Tool-assisted Verification of Java Programs*. PhD thesis, Radboud University of Nijmegen, 2006.
8. 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.
9. B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*. ACM Press, to appear, 2006.
10. 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.
11. R. Hähnle and W. Mostowski. Verification of safety properties in the presence of transactions. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Post Conf. Proc. of CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices, Marseille*, volume 3362 of *LNCS*, pages 151–171. Springer, 2005.
12. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.
13. 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*. Springer, 2002.
14. 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.
15. 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.
16. A. Platzer. Using a program verification calculus for constructing specifications from implementations. Master's thesis, Univ. Karlsruhe, Dept. of Computer Science, 2004.
17. P. Rümmer. Sequential, parallel, and quantified updates of first-order structures. In *Proceedings, 13th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, LNCS, 2006. To appear.
18. M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. The MIT Press, 1989.