# Combining Symbolic Execution and Model Checking to Reduce Dynamic Program Analysis Overhead

Néstor Cataño [*]

**Abstract**

This paper addresses the problem of reducing the runtime monitoring overhead for programs where fine-grained monitoring of events is required. To this end we complement model checking techniques with symbolic reasoning methods and show that, under certain circumstances, code fragments do not affect the validity of underlying properties. We consider safety properties given as regular expressions on events generated by the program. Further, we show how our framework can be extended to consider programs with cycles. We sample our presentation with the aid of the Java PathFinder model checker [13].

**Keywords**: model checking, Java PathFinder, symbolic reasoning, instrumentation, monitoring, invariant strengthening.

## 1 Introduction

Testing is a method to check the satisfaction of a property in an implementation by means of experimentation. In testing, test cases are designed following what the experience of programmers suggests. Unlike other more formal techniques such as model checking and theorem proving, testing is not complete in the sense that it can only prove the presence of errors but not their absence.

When doing testing, in general it is not possible to cover all the possible cases for which a program could produce an error, since that would imply, at least, considering a case for each possible value of each variable, but variable domains are usually infinite.

Model checking [1, 2] is a verification technique especially conceived to prove properties of reactive systems. When model checking, ideally a user would not need to interact with the model checker at all; the user's main work would consist in pressing the model checker "go-ahead" button, waiting a couple of seconds, and finally, analyzing the result produced by the model checker. In practice however, model checkers need human interaction. Additionally, model checking techniques do not scale well for real-life problems because of the the state explosion problem.

*Symbolic reasoning* [10] arises as a means to supplement testing and model checking. To supplement testing because, when reasoning symbolically, variables are not constrained to a concrete value but are supposed to have generic symbolic values. Hence, it is easier to cover more test cases. Symbolic reasoning supplements model checking techniques as well, because an explicit-state model checker can be used, for example, to explore symbolic trees. Model checking can thus be used more effectively to prove (as opposed to refuting) properties.

In the following, we formulate the problem which this paper is concerned with. We will then describe how model checking techniques, enhanced with symbolic reasoning, can be used to address the problem.

[*]Department of Computer Sciences, The University of York, U.K. `catano@cs.york.ac.uk`

**The problem.** We consider systems composed of two components. The first component, a Java program in execution, is "monitored" by the second, an external observer. Monitoring consists in tracking variables and their values. The observer is given as a finite-state automaton which verifies properties expressed as regular expressions on events generated by the program. In order for the observer to verify the property, the Java program needs to be instrumented to transmit variables and values. This instrumentation basically consists of many communication instructions. Since such emission (communication) instructions negatively affect the performance of the program, one is faced with the problem of reducing as many useless emissions as possible. An emission is considered to be useless when it does not modify the semantics of the observer. The semantics of the observer is expressed as "the ability of transition under the same program conditions".

Here we only consider safety properties. For example, one might like to monitor that "the temperature never exceeds 100 degrees", where the temperature is given by variable `temp` in the program; hence, the value of `temp` will be emitted to the observer whenever `temp` is updated and the observer will make a transition when `(temp > 100)` is `true`.

**This paper.** We use model checking techniques to address the problem of reducing the number of such useless emissions for programs where fine-grained monitoring of events is required. We define fine-grained monitoring as those cases where many statements in the program can affect the observations, for example, in the case where the specific value of a variable is tracked. To monitor a certain variable `x`, instructions `emit(x)` must be introduced in the Java program in any place where `x` is modified; `emit(x)` communicates the variable `x` and its current value to the observer. Although we will focus our presentation on fine-grained monitoring, techniques proposed here will also work for larger-scale monitoring, but will not have the same degree of effectiveness.

We propose the use of model checking to show that for certain code fragments, under certain conditions, no emission will be required since it cannot affect the property that is checked. Specifically, we will show how to determine whether program statements will change the state of some observer, by doing a symbolic execution of the code within the Java PathFinder (JPF) model checker [13]. JPF has been recently extended with the capacity of doing symbolic reasoning [8], and it is this feature that we will use here to show under which conditions a program statement can change the state of an observer.

In some cases, model checking can show conclusively that a program will behave correctly according to some property, *i.e.* any execution of the program will be correct. However, model checking is in general much better at finding counterexamples than showing that a program is correct. Since our approach relies on the model checker being able to show that a property holds, rather than showing a refutation, we believe that symbolic reasoning is more appropriate than model checking based on explicit enumeration.

Since our approach essentially starts from the assumption that every statement potentially changes the state of the monitoring system, with model checking then used to reduce this number of observations, the scalability of model checking influences the accuracy and not the correctness of our approach. In other words, the more analysis is done with the model checker, the less runtime overhead during monitoring.

**Contributions.** This paper shows how model checking techniques — when supplemented with symbolic reasoning methods — can effectively be used to show that a certain property holds. The major weakness of the symbolic execution within JPF is that cycles cannot be handled in their full generality: when doing symbolic execution, JPF cannot determine whether a state has been revisited, and therefore the analysis will not terminate. To overcome this, we show how classical reasoning about loop invariants can be used within our framework to deal with cycles. Roughly speaking, when a loop invariant is known, emissions are added to the program only when they do not contradict the invariant. The other emissions are useless and therefore should not be added.

Although, in general, finding loop invariants is an undecidable problem, we show how symbolic reasoning can be used to show that a property is a loop invariant. To do so, a loop invariant is conjectured — it is assumed at the beginning of the loop-body, and then checked immediately before its end — and successively refined, based on the information given by the symbolic execution of the loop.

**The rest of this chapter.** The rest of the chapter is structured as follows. Section 2 formalizes the problem of removing useless emissions when instrumenting programs in our framework. Section 3 introduces symbolic execution of programs. Ideas presented in this section provide a theoretical basis for a better understanding of subsequent sections. Section 4 gives an overview of the symbolic execution framework built on top of JPF. Section 5 presents how model checking with symbolic reasoning can be used to show the existence of useless emissions of non-looping programs. Section 6 extends Section 5 to consider loops. Lastly, Section 7 concludes and compares to related work.

## 2 Semantics of the observer

When running, a program generates events that can produce changes in the current state of the observer. We are interested in fine-grained monitoring, so any change of a specific variable value can potentially affect the semantics of (the current state of) the observer. Hence, only if each variable is tracked and its value sent to the observer (who will lastly check the property), the monitoring will be effective. To achieve an effective tracking, we instrument programs in such a way that emission statements are added in those parts of the code where modifications of variables are produced. These emission statements transmit the current value of the involved variables.

Emissions are "expensive", so having many of them will affect the performance of the program. We wish to remove as many useless emissions as possible. Useless emissions are those that, regardless of the transmitted values, will not make the observer transition.

To be tracked, instructions must be associated to locations. This association is only possible for non-looping programs. For looping programs this association cannot be done because in the general case the number of loop iterations cannot be decided without executing the loop itself. Section 6 presents our approach in dealing with loops. The rest of this section defines in detail the semantics of the observer.

**Instructions and program locations.** We call `i(loc)` the program instruction occurring at the program location `loc`. In the program below, method `m` declares a sole variable `x` which is then increased four times. Locations have been added into the program as comments; `i(0)` makes allusion to the instruction declaring the variable `x`, and successive instructions increasing variable `x` are referred to as `i(1)` through `i(4)`. The set of program locations is called `L`.

```
static void m() {
 0: int x = random();
 1: x++;
 2: x++;
 3: x++;
 4: x++;
}
```

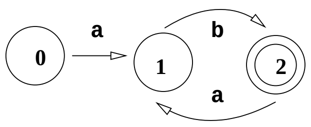


Figure 1: Automaton for $\texttt{(ab)}^{+}$

**Instrumentation.** Program instrumentation is accomplished by adding a single emission statement $\texttt{emit}(\texttt{loc}, \overrightarrow{\texttt{x}})$ after each instruction `i(loc)`, where $\overrightarrow{\texttt{x}}$ represents the set of variables involved in the execution of the instruction. Emission $\texttt{emit}(\texttt{loc}, \overrightarrow{\texttt{x}})$ sends the observer each variable in $\overrightarrow{\texttt{x}}$ as well as its value.

**Event generation.** When a program is executed, it generates events that can make the observer transition. We define events generated by programs as having type $\texttt{Ep} : \texttt{Vr} \times \texttt{Vl}$ between variables `Vr` and values `Vl`, with the intuitive meaning "if one is interested in events as described by the relational operator `Ep`, and after the execution of a certain instruction the value of variable `x` becomes `v`, then the event as described by `Ep(x,v)`[1] is produced by the execution of the instruction".

The event generation relation $\texttt{Evt} : \texttt{L} \to \mathcal{P}(\texttt{Ep})$ associates a location `loc` with the set of events the instruction at location `loc` is able to produce. In our program, the event generation relation `Evt` depends on the value taken by `x` in its declaration. For instance, if `x`'s initial value is 0 then $\texttt{Evt}(0) = \{\texttt{x} = 0\}$, $\texttt{Evt}(1) = \{\texttt{x} = 1\}$, $\texttt{Evt}(2) = \{\texttt{x} = 2\}$, $\texttt{Evt}(3) = \{\texttt{x} = 3\}$ and $\texttt{Evt}(4) = \{\texttt{x} = 4\}$.

**Observer.** We consider safety properties only, which are given as regular expressions over events tracked by the observer. Due to the relation between regular expressions and finite-state automata we chose these last as a model for the observer.

The observer is represented by the automaton $\mathcal{A} = (\texttt{Q}, \texttt{F}, \texttt{q}_0, \texttt{E}, \delta, \texttt{Mp})$, where `Q` is the set of states of the automaton, `F` its set of final states, $\texttt{q}_0$ its initial state, `E` its alphabet (of events), and $\delta : (\texttt{Q} \times \texttt{E}) \to \texttt{Q}$ its transition relation. Instructions in the program generate a spectrum of events that should be mapped into words as understood by the automaton. Therefore, the automaton $\mathcal{A}$ is provided with a mapping relation $\texttt{Mp} : \texttt{Ep} \to \texttt{E}$.

As an example, the automaton observer in Figure 1 is derived from the regular property $\texttt{(ab)}^{+}$, and its alphabet of events `E` is the set $\{\texttt{a},\texttt{b}\}$. Transitions for events other than `a` and `b` are undefined, *i.e.* they are supposed to be going into a certain trapping state. A mapping relation `Mp` for this automaton might, for instance, associate `x=1` in the program to event `a` in the observer; likewise `x=3` to `b`.

**Semantics.** We define two relations on an automaton $\mathcal{A}$=$(\texttt{Q},\texttt{F},\texttt{q}_0,\texttt{E},\delta,\texttt{Mp})$. The reaction relation $\texttt{React} : \texttt{L} \times \texttt{Q} \to \mathcal{P}(\texttt{E})$ relates a location `l` in the program and a state `q` in the automaton to the set of events $\texttt{e} \in \texttt{Evt}(\texttt{l})$ for which $\delta(\texttt{q}, \texttt{e}) \neq \texttt{q}$. Also, relation $\texttt{Stay} : \texttt{L} \times \texttt{Q} \to \mathcal{P}(\texttt{E})$ is defined as the `React` complement relation, found when considering elements $\texttt{e} \in \texttt{Evt}(\texttt{loc})$ for which $\delta(\texttt{q}, \texttt{e}) = \texttt{q}$ or $\delta(\texttt{q}, \texttt{e})$ is *undefined*.

If variable `x` is initialized to 0, and `Mp` is the same mapping as before, the automaton will react (make a transition) with event `a` each time the current state of the automaton is 2 and the program is at location 1 — `React(1,2)(a)`[2] holds. In contrast, when the automaton is at state 1 and the program at location 1 no reaction will be produced, *i.e.* `Stay(1,1)(a)` holds.

[1]Henceforth the more comfortable infix notation `x Ep v` will be used instead.
[2]Notice that sets are just predicates, so $\texttt{s} \in \texttt{S}$ is equivalent to `S(s)`.

| | | |
|---|---|---|
| Procedure | ::= | `procedure` Id( $\overrightarrow{param}$ ) Body `endp;` |
| Body | ::= | DeclS StmtS |
| DeclS | ::= | Decl DeclS |
| | \| | Decl |
| Decl | ::= | `declare` Vars : Type ; |
| | \| | ; |
| StmtS | ::= | StmtStmtS |
| | \| | Stmt |
| Stmt | ::= | Assig |
| | \| | IfStmt |
| | \| | WhileStmt |
| | \| | ProcInv |
| | \| | Return |
| | \| | ; |
| Assig | ::= | Var := Exp |
| IfStmt | ::= | `if`(Cond) $\mathsf{Body}_1$ `else` $\mathsf{Body}_2$ |
| WhileStmt | ::= | `while`(Cond) `do` Body |
| ProcInv | ::= | Id( $\overrightarrow{actual}$ ); |
| Return | ::= | `return`(Exp) |
| | \| | `return` |

Figure 2: A simple Java imperative language

For any location $\mathtt{l} \in \mathtt{L}$, if for all $\mathtt{q} \in \mathtt{Q}$ and for all $\mathtt{e} \in \mathtt{E}$ $\mathtt{Stay}(\mathtt{l}, \mathtt{q})(\mathtt{e})$ holds, then emission at location `l` is useless. We want to remove all those useless emissions. We propose the use of model checking and symbolic execution to show that under certain conditions these removals are always possible.

**Syntax for our programs.** Figure 2 introduces the syntax for our programs; this syntax is defined in a Java-like style. In the declaration of a procedure (nonterminal Procedure), TypeRtrn is the type of the expression returned by the procedure (a `boolean`, an `integer` or the special symbol `void` meaning no value), Id the name of the procedure and $\overrightarrow{param}$ the list of its parameters. As usual in Java-like programs, after declaring the procedure signature, one goes on to the variable declaration section (Decl in rule for Body) and then continues writing the procedure statements (nonterminal StmtS). Those declared variables are exclusively bounded to the Body of the procedure.

A statement Stmt can take the form of a variable assignment, an `if` statement, a `while` statement, a procedure invocation or simply a skip instruction "`;`", doing nothing. An assignment such as:

Var := Exp

assigns the value of the expression Exp to the variable Var. Integer expressions are formed with the aid of the usual binary arithmetic operators `+`, `-`, `*` and `/`, and the unary arithmetic operator `-`. Boolean expressions `Cond` are constructed from Boolean constants `true` and `false`, and from arithmetic expressions connected by relational operators `=`, `!=`, `<`, `<=`, `>`, or `>=`, and logical operators `&` (and), `|` (or), `!` (negation) and `=>` (implies).

An `if` statement as the following:

$$\texttt{if(Cond) Body}_1\ \texttt{else Body}_2$$

executes either $\texttt{Body}_1$ or $\texttt{Body}_2$ depending on the truth value of `Cond`. Finally, the procedure invocation:

$$\texttt{Id(}\overrightarrow{actual}\texttt{);}$$

causes the procedure `Id` to be invoked with parameters $\overrightarrow{actual}$. Formal parameters $\overrightarrow{param}$ in the declaration of the procedure `Id` are replaced by actual parameters, *i.e.* $[\overrightarrow{actual}/\overrightarrow{param}]$, using the `Java` convention.

## 3 Symbolic execution of programs

The symbolic execution of a program goes through symbolic states. A symbolic state is a tuple $(\texttt{x}_i\texttt{:X}_i\texttt{;pc:Bool})$ composed of symbolic variables $\texttt{X}_i$ for variables $\texttt{x}_i$, and a so-called *path condition* `pc`. The path condition is a quantifier-free boolean formula over symbolic inputs, that accumulates constraints which inputs must satisfy in order (for an execution) to follow the particular associated path. The initial path condition, *i.e.* the path condition of the initial symbolic state, is `true` for any program. The path condition is updated as more program instructions are executed. Further, a path condition is not allowed to be `false` (an unreachable path). The meaning of the symbolic execution of programs is defined as follows:

(*i.*) *Symbolic value of expressions.* Given `x:X` and `y:Y` in some symbolic state, the symbolic value of the expression `x op y` is `X op Y`, where `op` is any of `+`, `-`, etc.

(*ii.*) *Symbolic execution of assignments.* The symbolic execution of an assignment `x:= Exp` updates the symbolic state `(x:X;pc)` to the state `(x:Symb(Exp),pc)`, where `Symb(Exp)` is the symbolic evaluation of expression `Exp` as described by Item *(i.)*.

(*iii.*) *Symbolic execution of conditional statements.* The symbolic execution of a conditional statement $\texttt{if(Cond) Body}_1\ \texttt{else Body}_2$ is accomplished according to the following steps. *(a.)* First, evaluate the boolean expression `Cond`[3], *(b.)* If the current path condition `pc` implies `Cond`, then no new sub-cases is necessary because `pc` already contains enough information to deduce that $\texttt{Body}_1$ must be executed. If `pc` implies `!Cond` then similarly the path condition does not require to be updated because it already contains enough information to deduce that $\texttt{Body}_2$ must be executed, else *(c.)* Establish a sub-case where the path condition of the current symbolic state is changed from `pc` to `pc & Cond`, then proceed to symbolically execute $\texttt{Body}_1$, and *(d.)* Establish a sub-case where the path condition of the current path condition is changed from `pc` to `pc & !Cond`, then proceed to symbolically execute $\texttt{Body}_2$.

(*iv.*) *Symbolic execution of annotations.* To symbolically execute the input annotation `assume Exp ;` first evaluate `Exp`, *i.e.* `Symb(Exp)`, and then update the path condition `pc` to `pc & Symb(Exp)`. For output annotations `assert Exp ;`, first evaluate `Exp`. And, if `pc` implies `Symb(Exp)` then the program is correct, otherwise the program fails.

[3]Note that this requires counting on a decision procedure. We will not go further into this topic here; we just assume that this decision procedure exists.

## 4 Symbolic execution in Java PathFinder

Our symbolic execution-based framework uses the Java PathFinder model checker (JPF) [13]. JPF is an explicit-state model checker for Java programs built on top of a custom-made Java Virtual Machine (JVM). JPF can handle all the language features of Java, and additionally it treats non-deterministic choice expressed in annotations of the program being analyzed. For symbolic execution, the JPF model checker has been extended to allow backtracking whenever a path condition is unsatisfiable. To determine the satisfiability of a formula, JPF calls a decision procedure provided by the Omega library [12]. In particular, an annotation `ignoreIf(cond)` is used to allow backtracking: whenever `cond` evaluates to `true` the model checker will stop exploring the branch and backtrack. This feature will also be used in the discovery of loop invariants in Section 6.

The main idea behind symbolic execution [10] is to use symbolic values, instead of actual data, as input values, and to represent the values of program variables as symbolic expressions. The state of a symbolically executed program includes, in addition to the symbolic values of program variables, the program counter and a path condition. The path condition is a quantifier-free boolean formula over the symbolic inputs; it accumulates constraints which the inputs must satisfy in order for an execution to follow the particular associated path. A symbolic execution tree characterizes execution paths followed during the symbolic execution of a program. The nodes represent program states and the arcs represent transitions between states.

As an example (taken from [8]), consider the code fragment in Figure 3, which swaps the values of integer variables `x` and `y`, when `x` is greater than `y`. Figure 3 also shows the corresponding symbolic execution tree. Initially, the path condition, `PC`, is `true` and `x` and `y` have symbolic values `X` and `Y`, respectively. At each branch point, `PC` is updated with assumptions about the inputs according to the possible alternative paths. For example, after the execution of the first statement, both `then` and `else` alternatives of the `if` statement are possible, and `PC` is updated accordingly. If the path condition becomes `false`, *i.e.* no set of inputs satisfy it, this means that the symbolic state is not reachable, and the symbolic execution does not continue on that path. For example, statement `(6)` is unreachable.

Symbolic execution techniques have traditionally come up in the context of sequential programs with a fixed number of integer variables. In [13], these techniques have been extended to handle dynamically allocated data structures (lists and trees), complex preconditions (disallowing cyclic lists), other primitive data (strings), and concurrency. A key feature of the algorithm implemented in JPF is that it starts the symbolic execution of a procedure on *uninitialized* inputs and uses *lazy initialization* to assign values to these inputs. Consequently, the parameters are initialized when they are first accessed during the symbolic execution of the procedure. This allows symbolic execution of procedures without requiring an *a priori* bound on the number of input objects. Procedure preconditions are used to initialize inputs with valid values only.

**Recursion.** The JPF algorithm implementation exploits the model checker's search capabilities to handle arbitrary program control flow. In the implementation, no requirement on the model checker to perform state matching is done since state matching is undecidable when states represent path conditions on unbound data. Furthermore, the symbolic execution of looping programs can explore infinite execution trees. To overcome this, Section 6 describes how to address cycles in Java PathFinder.

```
    int x, y;
    read x,y;
1:  if (x > y) {
2:    x = x + y;
3:    y = x - y;
4:    x = x - y;
5:    if (x > y)
6:     assert(false);
    }
```

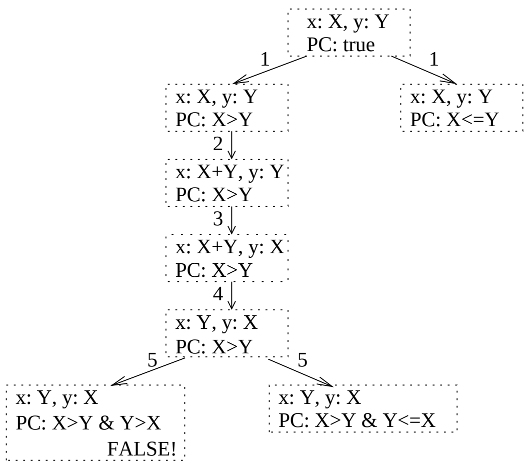


Figure 3: Code for swapping integers and corresponding symbolic execution tree.

## 5 Eliminating useless emissions for Java sequential programs

We use model checking and symbolic reasoning to show that we can remove useless emissions from a program and still *preserve* the semantics of the observer. The observer's semantics is defined as "the ability of reacting under the same program conditions", and formalized by the predicate `React` in Section 2.

Consider the instrumented program below, where an emission statement has been added into the program immediately after each modification of variable `x`, the only program variable. Also, consider the observer for the property $\texttt{(ab)}^+$ in Figure 4, and the event mapping relation `Mp` which associates the event `x=1` in the program to the event `a` in the automaton, and `x=3` to `b`. The automaton in Figure 4 only reacts when variable `x` is 1 or 3. Variable `x` is initially given a nonnegative value as returned by function `random`; after the fourth increment of `x`, the value will be greater or equal than 4. Hence, the last emission will produce no reaction in the automaton, *i.e.* `Stay(4,q)(a)` for any $\texttt{q} \in \texttt{Q}$. Therefore, this emission becomes useless and it can be removed from the instrumented program.

```
static void m() {
 0: int x = random(); emit(0,x);

 1: x++; emit(1,x);
 2: x++; emit(2,x);
 3: x++; emit(3,x);
 4: x++; emit(4,x);
}
```

Below we present how the symbolic version of the program above is first elaborated and then model checked in the symbolic execution framework of JPF. We show that the last emission can be removed from the instrumented program, while preserving the semantics of the observer. Variable `x` of type `int` has been replaced by variable `X` of type `SymbolicInteger`, an integer implementation for `Expression`. Classes `SymbolicInteger` and `Expression` are defined in the module for symbolic execution built on top of JPF [8].

```
static void m() {
 0: Expression X = new SymbolicInteger();
    _addDet(GE,X,0);
```

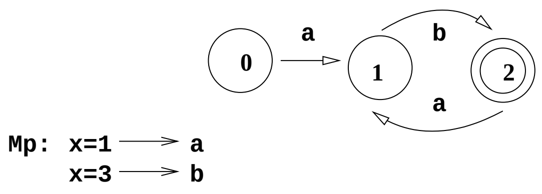


Figure 4: Mapping for `(ab)`$^+$

```
    Emit(0,X);

 1: X = X._plus(1); Emit(1,X);
 2: X = X._plus(1); Emit(2,X);
 3: X = X._plus(1); Emit(3,X);
 4: X = X._plus(1); Emit(4,X);
}
```

After `X` is created, the condition `_addDet(GE,X,0)` saying that "`X`'s values are always greater or equal than 0" is added[4]. Moreover, operations increasing `x` have been replaced by method calls to `X`, *i.e.* `X._plus(1)`.

We also need to create a symbolic version for emissions. This symbolic version takes into account the way the automaton evolves from a particular state to another when the program is executed. Since each emission is associated with a location, the symbolic version `Emit` must be parameterized by the location.

The coding below presents the symbolic emission function `Emit` as it is implemented in JPF. The variable `state` in the guard of the `if` statement represents the automaton's current state, and the `_add` instructions reflect the behavior of the automaton's transitions. When executing the JPF model checker on the whole symbolic program, and once the conditions on the guard of the `if` statement are verified, the new conditions on `X` are added to the path condition. These `_add` instructions encode all possible reactions of the automaton. Hence, when the automaton stays in the same state, no new condition on `X` is added. Method `simplify` returns the path condition on `X` necessary for reaching the program location `loc` from the starting condition `x=x_init`[5]. Finally, method `changeState` updates the current state of the automaton according to the current path condition on `X`.

```
static void Emit(int loc,Expression X) {
 if(((state==0)&_add(EQ,X,1)) ||
    ((state==1)&_add(EQ,X,3)) ||
    ((state==2)&_add(EQ,X,1))) {
   simplify(Expression.pc,loc,X);
   changeState(X);
 }
}
```

When the whole program is symbolically executed in Java PathFinder, the path conditions `x=1`, `x=0`, `x=1`, `x=0` and `false` for respective locations 0 to 4 are yielded. From the last condition we conclude that the last emission will produce no reaction on the automaton for any `x`'s initial value. Therefore, that emission can be removed, while the semantics of the observer stays unchanged.

We use all those conditions on `x` not only for removing the last emission but for executing the other emission statements under the conditions yielded at each respective location. The

[4]This reflects the fact that `random` in the original program always returns a nonnegative integer value.

[5]`x_init` is the initial value returned by `random`.

final instrumentation for method m is presented below, where the previous conditions on variable x have been integrated into the original program. The last emission will never happen. Notice that all these conditions refer to the initial value x_init of x.

```
public static void m() {
 int x = random();
 int x_init = x;

 if (x_init == 1) emit(x);

 x++; if (x_init == 0) emit(x);
 x++; if (x_init == 1) emit(x);
 x++; if (x_init == 0) emit(x);
 x++; if (false) emit(x);
}
```

Similar considerations can be made when dealing with the whole syntax presented in Figure 2, except for loops. Therefore, Section 6 extends the work presented in this section to consider loops.

## 6 Eliminating useless emissions in cycles

When considering cycles, a similar analysis as in Section 5 cannot be done: when doing symbolic execution, the Java PathFinder model checker cannot determine whether a state has been previously visited; also, because the number of loop iterations cannot be determined beforehand, no full mapping between locations inside the loop and emission statements can be performed.

Hence, we should do a clever analysis of the information at hand. For instance, if we know the loop invariant, we can decide to emit only in those cases when the conditions under which emissions will occur do not contradict the invariant. However, the main difficulty is that finding loop invariants is an undecidable problem. One can also work with *partial* information, *i.e.* one can symbolically execute the program for a fixed number n of iterations, conjecture a *partial invariant*, and then remove any emission contradicting this partial invariant. Obviously, the intrinsic problem is that this partial invariant can be invalidated by a subsequent loop iteration.

In the following, we show how symbolic execution can be used to show that an initial conjectured invariant is a real invariant, and then, how to use this invariant information to remove emissions that produce no reaction in the automaton.

**Guessing an initial conjectured invariant.** First, a (symbolic) loop program is created for which all variables controlling the way the loop iterates are symbolic. Then, this (partially) symbolic program is executed and the path conditions generated at each loop iteration are checked. We conjecture a loop invariant based on the partial information at hand and proceed to prove that this conjecture is a real invariant.

We have rewritten "in a loop style" the example presented in Section 5 (see below). The only difference resides in the fact that we now have a random number n of assignments to x instead of a fixed number of 4 iterations. Furthermore, variable i has been introduced to control the current loop iteration.

```
public static void m() {
 int x = random();
 emit(0,x);
 int n = random();
 int i = 0;
 while(i < n) {
  i++;
  x++; emit(i,x);
 }
}
```

The program below is the symbolic on `X` version of the program above, while variables `i` and `n` remain concrete. We want to execute this symbolic program for different values of `n` in order to conjecture a loop invariant. We start from `n=6`.

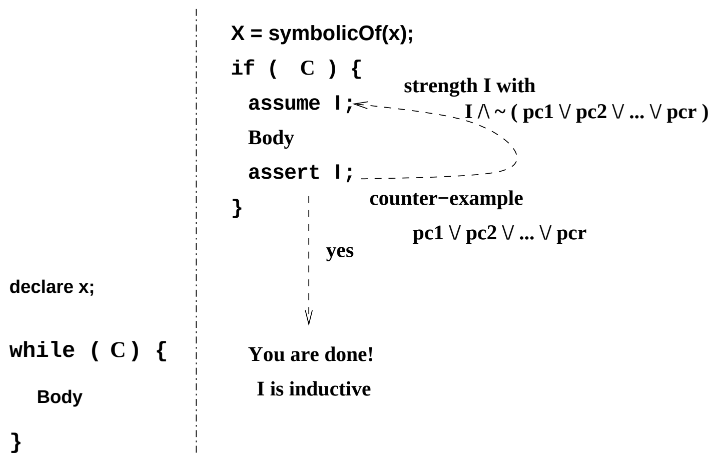
```
public static void m() {
 Expression X = new SymbolicInteger();
 _addDet(GE,X,0);
 Emit(0,X);

 int n = 6, i = 0;
 while(i < n) {
  i++;
  X = X._plus(1); Emit(1,X);
 }
}
```

When running JPF on this program, the path conditions `(x=1 || x=3)` for emission at `i=0`, `(x=0)` for emission at `i=1`, `(x=1 || x=2)` for emission at `i=2`, and `x=0` for `i=3` are established. Of the three remaining emissions for the three remaining iterations none is performed. Then, the number of iterations is increased to `n=13`. This time, the previous conditions for location `0` and for the six first iterations of the loop remain the same. No emission is performed for the other `7` iterations. Hence, we conjecture the invariant *"no emission is performed once* `i` *becomes greater than 3"* and proceed to prove whether it is a real invariant.

**Proving the conjectured invariant.** In general, symbolic execution of looping programs can produce *infinite* symbolic trees because another new path condition might be added each time the loop guard is visited. Hence, if symbolic techniques are to be used to prove that an initial conjecture $I_k$ is a loop invariant, then the looping program must be made non-looping, *i.e.* the looping program must be *cut.* This non-looping program, however, must be general, in the sense that it must be representative of any symbolic execution visiting the loop guard. To achieve this generality, loop-guard variables must be declared symbolic, and then, when executed symbolically, their path condition initialized to the most general condition, namely, `true`.

The `if` program in Figure 5($b$) is obtained when the symbolic tree of the `while` program in Figure 5($a$) is cut. This new program provides a convenient way to show that the initially conjectured property $I_k$ is an invariant. To show that $I_k$ is an invariant, $I_k$ must be *assumed* immediately after the guard, and then *asserted* immediately before the end of the body. If the evaluation of that assertion succeeds, then $I_k$ is inductive. Otherwise, if the evaluation fails, the information given by the symbolic program is employed to strengthen $I_k$: if the execution of `assert` $I_k$; fails producing path conditions $pc_1 \vee \ldots \vee pc_k$, then $I_k$ is strengthened to $I_{k+1} = I_k \wedge \neg(pc_1 \vee \ldots \vee pc_k)$, and the whole process is restarted; this time from $I_{k+1}$.

Below we present the full symbolic loop program produced when JPF is used through this iterative process of strengthening and checking. From Lines `2:` to `4:` symbolic variables for every concrete variable — including those concerned with the loop iterations — are created. From Lines `5:` to `7:`, initial conditions for these symbolic variables are added, *e.g* "`X` is



(a) `while` loop (b) Iterative invariant strengthening

Figure 5: Loop invariant strengthening

nonnegative". When an `if` statement is executed symbolically, the guard must be asserted as a precondition to the execution of the loop body, Line `15:`. Also, the initial conjecture has been incorporated as a necessary condition to emitting, Line `18:`. Each one of Lines `9:` to `13:` represents an iteration refinement of the initial conjecture. Each one of these refinements are again checked after the loop body has been executed, Lines `20:` to `25:`. The last iteration produces no path condition invalidating the precedent strengthened $\mathtt{I_k}$. Hence, the loop invariant is the conjunction between the initial conjecture and each refinement.

```
 1:public static void m() {
 2: Expression X = new SymbolicInteger(),
 3:            N = new SymbolicInteger(),
 4:            I = new SymbolicInteger();
 5: _addDet(GE,X,0);
 6: _addDet(LT,I,N);
 7: _addDet(GT,I,0);

 9: ignoreIf(_add(GE,I,3)&_add(EQ,X,0));
10: ignoreIf(_add(GE,I,3)&_add(EQ,X,2));
11: ignoreIf(_add(GE,I,2)&_add(EQ,X,1));
12: ignoreIf(_add(EQ,I,1)&_add(EQ,X,0));
13: ignoreIf(_add(EQ,I,2)&_add(EQ,X,0));

15: _addDet(LT,I,N);
16: I = I._plus(1);
17: X = X._plus(1);
18: if (_add(GT,I,3)) assert(false);

20: if((_add(GE,I,3)&_add(EQ,X,0)) ||
21:    (_add(GE,I,3)&_add(EQ,X,2)) ||
22:    (_add(GE,I,2)&_add(EQ,X,1)) ||
23:    (_add(EQ,I,1)&_add(EQ,X,0)) ||
24:    (_add(EQ,I,2)&_add(EQ,X,0))
25:   ) assert(false);
26:}
```

Before being sure of whether this invariant is really inductive, a further condition must be checked. The invariant should be valid at the initial state. This condition must be checked in the instrumented program before the loop can be executed.

The instrumented program is shown below. From Lines `6:` to 8 the initial invariant condition is checked. For this particular example, this condition will always hold because the initial value for `i` is 0. In Line `12:`, the statement `emit(i,x)` is executed under the initial conjecture condition. Only when the condition evaluates to `true`, `x`'s value will be emitted. For the other cases we can not emit and still preserve the semantics of the observer, *i.e.* if `i>3`, then `Stay(i,q)(a)` and `Stay(i,q)(b)` for any `q` in the observer's set of states.

```
 1:public static void m() {
 2: int x = random(),
 3:     n = random(),
 4:     i = 0;

 6: if(((i>=2)&(x==0))||((i>=3)&(x==2))||
 7:    ((i>=2)&(x==1))||((i==1)&(x==0)))
 8:   println(``Invariant broken'');

10: while(i<n) {
11:  i++;
12:  x++; if(!(i>3)) emit(i,x);
13: }
14:}
```

## 7 Conclusion and related work

In this paper we showed that model checking techniques can be used to reduce dynamic program analysis overhead. Our approach basically consists in proving that certain code fragments will generate no reaction on the automaton-based monitoring system, and hence these code fragments can be removed. We developed our work in the framework of the Java PathFinder model checker (JPF), although the work is still valid in the framework of any model checker doing symbolic reasoning.

Our approach is general in the sense that there is no *a priori* restriction on the way the mapping from program events into events as understood by the observer is performed. However, when employing JPF to do symbolic execution, considered (automaton-based) properties cannot include transitions on the same state. This is not a drawback of our approach, but an aspect where the JPF model checker can be improved.

A drawback of the approach presented in this paper is that the refinement process is not always convergent and hence, it could happen that an invariant is never obtained, even though the invariant might exist. The whole process depends on the initial conjectured invariant and on the way each successive strengthened $\mathtt{I}_{\mathtt{k+1}}$ is calculated.

*C. Pasareanu* and *W. Visser* in [11] propose an heuristic to achieve termination in this iterative process of strengthening. Yet, their problem is slightly different to that presented here. They are interested in showing that a loop program respects some property `P`[6]. They use an invariant strengthening algorithm similar to that described before, but additionally, at each step $k$ of strengthening, the "exact" invariant $I_k$ is newly strengthened iteratively. In this new strengthening process, the effect produced by the assertion of $I_k$ at the end of the loop-body is not considered. If an error exists, then the method is guaranteed to terminate. If the program is correct with respect to the property, the method might not terminate.

[6]That is, `assert P;` is added immediately after the loop statement.

In [6, 7], *K. Havelund* and *G. Rosu* report on runtime verification in the framework of Java PathExplorer. It is composed of three main modules, namely, an instrumentation module, an interconnection module, and an observer module. The instrumentation module modifies the program byte codes so that relevant emissions are sent to the interconnection module, which in turn will retransmit these events to the observer module. The observer module checks for validity of temporal properties. Ideally, techniques to reduce program analysis overhead presented in this paper would serve as basis to enhance the runtime verification labor done by Java PathExplorer.

Two tools close to Java PathExplorer are Java-MaC (*M. Kim*, *S. Kannan*, *I. Lee* and *O. Sokolsky* in [9]) and DynaMICs (*A.Q. Gater*, *S. Roach*, *O. Mondragon* and *N. Delgado* in [5]). Java-MaC separates monitoring task from checking task. This separation makes Java-MaC an extensible open architecture. Unlike Java PathExplorer, in DynaMICs, properties targeted for verification are expressed as constraints. For instance, for the problem of the division of two integers `x` and `y`, yielding a quotient `q` and a reminder `r`, two constraints can be defined: `r < y` and `(q` $\times$ `y) + r = x`.

*C. Flanagan* and *Sh. Qadeer* in [4] present an abstraction-based method to infer loop invariants. They infer loop invariants to verify programs that manipulate unbound data such as arrays. Loop invariants for each loop are computed by iterative approximation. The problem of their approach is that it requires ingenuity in finding the initial property for iterative approximation.

*T. Colcombet* and *P. Fradet* in [3] propose a method to enforce trace properties. The programmer specifies the property `T` separately from the program `P`, and a "transformer" takes `T` and `P` and produces another equivalent program that satisfies the property. They only consider safety properties. An advantage of the work presented in this paper is that we consider not just "plain" events, but also values of variables and symbolic constraints over these variables.

As future work we plan to formalize and prove theorems to establish precisely the kind of programs that can be monitored with the methodology presented in this paper.

**Acknowledgements.** Thanks to Dr. Willem Visser for inviting me to NASA Ames in the summer of 2002, when this work was mainly carried out, and discussing topics on model checking and symbolic methods. Thanks to Dr. Corina Pasareanu with whom I had fruitful discussions on automatic loop-invariant generation.

## References

[1] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit L. Petrucci, Ph. Schnoebelen, and P. Mackenzie. *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer-Verlag, 1999.

[2] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 2000.

[3] T. Colcombet and P. Fradet. Enforcing trace properties by program transformation. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 54–66, 2000.

[4] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 191–202. ACM Press, 2002.

[5] A.Q. Gates, S. Roach, O. Mondragón, and N. Delgado. DynaMICs: Comprehensive support for run-time monitoring. In K. Havelund and G. Rosu, editors, *Electronic Notes in Theoretical Computer Science*, volume 55. Elsevier, 2001.

[6] K. Havelund and G. Rosu. Java PathExplorer — a runtime verification tool. In *Proceedings of 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space, ISAIRAS'01*, Montreal, Canada, Jun. 18–22 2001.

[7] K. Havelund and G. Rosu. Synthesizing monitors for safety properties. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 342–356, 2002.

[8] S. Khurshid, C. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of TACAS03: Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *Lecture Notes in Computer Science*, Warsaw, Poland, Apr. 2003.

[9] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Java-MaC: a run-time assurance tool for Java programs. In K. Havelund and G. Rosu, editors, *Electronic Notes in Theoretical Computer Science*, volume 55. Elsevier, 2001.

[10] J.C. King. Symbolic execution and program testing. *Communications of ACM*, 19(7):385–394, 1976.

[11] C. Pasareanu and W. Visser. Verification of Java programs using symbolic execution and invariant generation. In *Proceedings of Model Checking Software: 11th International SPIN Workshop*, Lecture Notes in Computer Science, Barcelona, Spain, Apr. 1-3 2004. Springer-Verlag.

[12] W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 31(8), Aug. 1992.

[13] W. Visser, K. Havelund, G. Brat, and S.J. Park. Model checking programs. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, Grenoble, France, Sept. 2000.