Combining symbolic execution and model checking to reduce dynamic program analysis overhead

Néstor Cataño

▶ **To cite this version:**

Néstor Cataño. Combining symbolic execution and model checking to reduce dynamic program analysis overhead. RR-5263, INRIA. 2004, pp.31. inria-00070735

HAL Id: inria-00070735
https://inria.hal.science/inria-00070735

Submitted on 19 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Combining symbolic execution and model checking to reduce dynamic program analysis overhead

Néstor Cataño

N° 5263

12 Juillet 2004

Thème SYM

ISSN 0249-6399 ISRN INRIA/RR--5263--FR+ENG

INRIA
SOPHIA ANTIPOLIS

Combining symbolic execution and model checking to reduce dynamic program analysis overhead

Néstor Cataño*

Thème SYM — Systèmes symboliques
Projet Everest

Rapport de recherche n° 5263 — 12 Juillet 2004 — 31 pages

Abstract: By using symbolic execution techniques in the framework of the Java PathFinder model checker, we show that some Java program statements are unable to change the current state of certain *offline observer*, and so it is useless to instrument them. The observer is given as a finite-state automaton and we considered safety properties.

Key-words: model checking, symbolic execution, Java, Java PathFinder, instrumentation, invariant strengthening, program correctness

* Nestor.Catano@sophia.inria.fr

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

Sur l'utilisation de l'exécution symbolique et du model checking pour réduire des émissions inutiles

Résumé : Dans le cadre de Java PathFinder, nous utilisons des techniques d'exécution symbolique pour démontrer que certaines parties d'un programme Java ne peuvent pas changer l'état actuel d'un *observateur autonome*, et que par conséquence il s'avère inutile de les instrumenter. L'observateur est exprimé sous la forme d'un automate à l'état fini, et nous avons considéré des propriétés de sûreté.

Mots-clés : model checking, exécution symbolique, Java, Java PathFinder, instrumentation, renforcement d'invariants, correction de programmes

Testing is a method to check the satisfaction of a property in an implementation by means of experimentation. In testing, test suites 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, to consider a case for each possible value of each variable, whose domains are usually infinite.

Model checking [2, 1] is a verification technique especially conceived to prove properties of reactive systems. In theory, model checking is a "push-button" technique: ideally a user does not need to interact with the model checker at all; his main work consists in pressing the model checker "go-ahead" button, to wait a couple of seconds, and finally, to analyze the result produced by the model checker. In practice, however, model checkers need of human interaction, and furthermore model checking techniques do not scale well for real-life problems because of the state explosion problem.

Symbolic reasoning [13] rises as a means to supplement testing and model checking. To supplement testing because when reasoning symbolically one disregards the specific value of a variable, and rather reasons in terms of generic symbolic values for expressions. Thus, one is sure of covering all the cases. Symbolic reasoning supplements model checking techniques as well, because one can for example use a model checker to generate and explore program symbolic trees. Thus, model checking can be used more effectively to prove (as oppose to "to refute") properties.

In the following we formulate first the problem dealt with this report; then we describe how we use model checking techniques, enhanced with symbolic reasoning, to address it.

The problem. We consider systems composed of two components. The first component, a Java program in execution, is "monitored" by the second, an extern observer. Monitoring consists in tracking variables and their values. The observer is given as a finite-state automaton that verifies properties expressed as regular expressions on events generated by the program.

To allow the observer to verify, 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 affect negatively the performance of the Java program in execution, 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 can be expressed as "the capability of transition under the same program conditions".

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 transition when `(temp > 100)` is `true`.

This report. 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. Thus, to monitoring 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 the presentation on fine-grained monitoring, the 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 being 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 with the Java PathFinder (JPF) model checker [16]. JPF has recently been extended with the capability of doing symbolic reasoning [10], and it is this feature that we will use here to show under what 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 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 does not influence the correctness of our approach, only the accuracy. In other words, the more analysis is done with the model checker, the more reduction in runtime overhead during monitoring.

Contributions. We show 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 shortcoming of JPF, 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, one only emits in those cases when emissions do not contradict the invariant. The other emissions are useless and therefore can (should) be removed from instrumentation. 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. One starts by guessing a loop invariant; this property is assumed at the beginning of the loop-body, and then checked immediately before its end. We use the

information given by the execution of the whole loop program (including the assumption and checking instructions) to successively refine the initial guessed loop invariant.

Finally we describe how modular instrumentation of programs can be achieved in our framework.

Related work. In [7, 8], *K. Havelund* and *G. Rosu* report on runtime verification in the framework of Java PathExplorer. This tool conceives the runtime verification task as we do in this report. More concretely, Java PathExplorer is a runtime verification tool composed of three main modules, namely, an instrumentation module, an interconnection module, and an observer module. The instrumentation module carries out an automated instrumentation of the program, basically modifying the program byte codes in such a way that relevant emissions will be sent to the interconnection module, which in turn will retransmit these events to the observer module. The observer module will check for validity of *temporal* properties. Ideally, techniques for reducing program analysis overhead presented in this report 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 [11]) 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 target for verification are expressed as constraints. Thus, for example, 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 for verifying programs that manipulate unbound data such as arrays. Loop invariants for each loop are computed by interactive approximation. The problem of their approach is that they require 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 report is that we consider not just "plain" events, but also values of variables and symbolic constraints over these variables.

The rest of this report. The rest of the report is structured as follows. Section 1 formalizes the problem of removing useless emissions when instrumenting a program. Section 2 introduces symbolic execution of programs. Ideas presented in this section provide a theoretical basis for a better understanding of subsequent sections. Section 3 gives an overview of the symbolic execution framework in JPF. Section 4 presents how model checking with symbolic reasoning can be used to show the existence of useless emissions of programs without loops. Section 5 extends Section 4 to consider loops. For loops, one must consider the

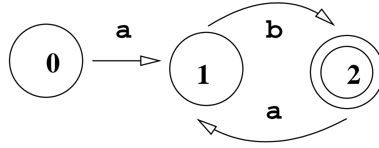


Figure 1: Automaton for `(ab)`$^+$

respective invariants. Section 6 shows how the work in Sections 4 and 5 can be carried out modularly. Lastly, Section 7 concludes.

1 Semantics for 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 value of a variable can potentially affect the semantics of (the current state of) the observer. Hence, for that monitoring is effective, any variable in the program must be tracked, and its value transmitted to the observer who ultimately will check the property.

To achieve an effective tracking, we instrument programs in such a way that emission statements are added in those code parts where (potential) 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 whole program in execution. Thus, 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 given locations. That allows one to reference those useless instructions. This mapping is possible only when programs do not containing loops. For program containing loops, it is not possible to decide in advance how many times it will iterate since that will require variables to be valued already. For loop programs, a different reasoning must be done as presented in Section 5.

The rest of this section defines precisely the semantics of the observer.

Instructions and program locations. Programs count on (program) locations. We call `i(loc)` the program instruction occurring at 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 referenced as `i(1)` to `i(4)`. The set of program instructions is called `I`, and the set of program locations `L`.

```
static void m() {
```

```
 0: int x = random();

 1: x++;
 2: x++;
 3: x++;
 4: x++;
}
```

Instrumentation. Programs are instrumented by adding an emission statement $\texttt{emit}(\texttt{loc}, \overrightarrow{x})$ after each instruction `i(loc)`, where $\overrightarrow{x}$ represents the set of variables involved in the execution of that instruction. The emission $\texttt{emit}(\texttt{loc}, \overrightarrow{x})$ sends the observer each variable in $\overrightarrow{x}$ together with its value.

Event generation. When executing programs, events are generated which can make the observer transition. We define events generated by programs as having the binary relation 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 binary operator `op`, and the execution of an instruction makes variable `x` gets value `v`, then the event as described by `op(x,v)`[1] was produced by the execution of that instruction".

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

Observer. We only consider safety properties, given as regular expressions over events tracked by the observer. Due to the relation between regular expressions and finite-state automata we chose this 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 $\delta : (\texttt{Q} \times \texttt{E}) \rightarrow \texttt{Q}$ its transition relation. Further, instructions in the program generate a spectrum of events that should be mapped into words as understood by the automaton. Hence, additionally, the automaton $\mathcal{A}$ is provided with a function $\texttt{Mp} : \texttt{Ep} \rightarrow \texttt{E}$ doing this mapping.

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

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

Semantics. We define two relations on an automaton $\mathcal{A}$=(Q,F,q_0,E,δ,Mp). The reaction relation React : L × Q → setof[E] relates a location l in the program and a state q in the the automaton with the set of events e ∈ Evt(l) for which $\delta(q, e) \neq q$. Relation Stay : L × Q → setof[E] is defined as the React complement relation, found when considering elements e ∈ Evt(loc) for which $\delta(q, e) = q$ or $\delta(q, e)$ is *undefined*.

If variable x is initialized to 0, and the same mapping Mp as above is given, the automaton will react (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) holds. In contrast, no reaction will be produced when the automaton is at state 1 and the program at location 1 — Stay(1,1)(a) holds.

For any location l ∈ L, if for all q ∈ Q and for all e ∈ E, Stay(l, q)(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 those removals are possible.

2 Symbolic execution of programs.

This section is intended to describe how symbolic execution can be used to prove correctness. To have a full description on symbolic execution and its theoretical underlying results, the reader is invited to consult [12, 13, 6].

This section. Ideas presented in this section provide a theoretical basis for a better understanding of the work carried out in Sections 4 and 5. Section 2.1 starts describing a simple program syntax on which we formulate later the concept of program correctness. This syntax allows one to write programs in a Java-like syntax, and although simple, it covers most important constructions we use when defining symbolic execution and program correctness. Sections 2.2 and 2.3 introduce symbolic execution and show how it can be used to show program correctness. Finally 2.4 extends the notion of program correctness to programs having loops.

2.1 A simple example syntax for our programs.

Figure 2 introduces the syntax for our programs; this syntax is defined in a Java-like simple style, so most of the constructs and their meanings are familiar for Java programmers. Procedures are declared with the aid of the keywords procedure and endp; indicating the begin and end of the declaration. After the keyword procedure, the Id represents the procedure name and $\overrightarrow{param}$ the list of parameters. As usual in Java-like programs, after declaring the procedure signature, one goes on to the variable declaration section Decl[3] (see rule for Body), and then one continues writing the procedure statements which do the work. Those declared variables are exclusively bounded to the Body of the procedure. Their type Type covers boolean and integer.

[3]Notice that, unlike Java programs, we force the declaration part to be placed at the beginning of the method.

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) Body_1 `else` Body_2
WhileStmt	::=	`while` (Cond) `do` Body
ProcInv	::=	Id ($\overrightarrow{actual}$) ;
Return	::=	`return` (Exp)
	|	`return`

Figure 2: Simple Java imperative language

The Body of the procedure is composed of a sequence of statements Stmt which can take the shape of a variable assignment, an `if` statement, a `while` loop, a procedure invocation, a procedure return statement, or simply a skip instruction "`;`" doing nothing.

An assignment such as

Var := Exp

assigns the value of the expression Exp to the variable Var. Expressions are formed with the help of the usual binary arithmetic operators `+`, `-`, `*` and `/`, and the unary arithmetic operator `-`; and Var is a variable declared in the scope of the procedure where used.

Boolean expressions Cond are constructed from the boolean constants `true` and `false`, and from arithmetic expressions connected by the relational operators =, !=, <, <=, >, or >=, and the logical operators & (and), | (or), ! (negation) and => (implies).

An `if` statement such as

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

executes either Body_1 or Body_2 depending on the truth value of Cond.

A `while` statement

`while(Cond) do Body`

executes Body as it was a single instruction.

A procedure invocation

$$\mathsf{Id}\,(\overrightarrow{actual})\,;$$

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

For convenience the initial value of a declared variable or a formal parameter is constructed with the help of the operator `pre`. For instance `pre(a)` for some parameter a refers to the value of a just after the invocation of the procedure.

A return statement

`return(Exp)`

causes the invoked procedure to return with the expression Exp as value. When no expression Exp is provided, the procedure just returns from the invocation. There exists a sole `return` statement for each procedure, which can be placed at any place in the procedure.

Program correctness. The idea of program correctness makes allusion to the proper "functional" behavior of a program, *i.e.* to the input-output behavior. Hence, one needs to count on a means to annotate what the inputs and outputs for a procedure are expected to be. These annotations constraint the functional behavior of the procedure. For stating inputs, we count on annotations of the form:

`assume Exp ;`

which is annotated at the beginning of the procedure, and makes Exp to hold upon procedure entry. If no `assume` annotation is provided for the procedure, the `assume true;` input annotation is assumed. Input annotations are placed immediately after the signature of a procedure declaration. Further, for output behavior we count on annotations of the form:

`assert Exp ;`

which fails if the boolean expression Exp cannot be established when encountered. Output annotations are placed immediately before the procedure `return` statement.

Below, on the left side, we declare a procedure that calculates the absolute value for a variable `x`. A first candidate for procedure output would be `y >= 0` which says that regardless of the value of `x`, the value returned by the procedure `AbsValue` will always be non negative. We can optionally add other outputs to achieve a more complete specification. For example, one could make explicit the intention of not modifying the parameter `x` by annotating the output `x = pre(x)`, or that the value returned by the procedure will be the parameter itself or its negation, *i.e.* `(y = pre(x) | y = -pre(x))`. The right side of the figure below shows the `AbsValue` procedure with annotations.

```
1: procedure AbsValue(x)        |     procedure AbsValue(x)
                                | 2:   assume (true)
                                |
3:   declare y : integer;       |      declare y : integer;
                                |
4:   if (x < 0)                 |      if (x < 0)
5:    then y := -x;             |       then y := -x;
6:    else y := x;              |       else y := x;
                                | 7:   assert (y = pre(x) | y = -pre(x))
                                |            & y >= 0 & x = pre(x)
8:    return y;                 |      return y;
```

In our previous notion of correctness we have not involved the notion of termination. One has two kind of correctness. First, *partial correctness* which indicates that a program behaves properly every time it terminates; and *total correctness* which additionally indicates that the procedure terminates for every possible input. We are concerned here with partial correctness hence no analysis of termination is done in the rest of this section.

2.2 Symbolic execution as a means of showing program correctness.

A correctness proof must be a proof for every possible input — in our example, every possible integer value for `x` — and all possible outputs. Because the possible values for procedure inputs and outputs are in general infinite, we cannot adopt a "sub-proof" approach consisting in making a proof for each input-output possible case. Instead, we might opt for reasoning in terms of symbolic values for the program variables and then to reason in terms of certain cases for those general variables. It is this feature which makes symbolic reasoning a convenient technique to show program correctness. In the following, we describe a correctness proof for the annotated program `AbsValue` using symbolic reasoning, then formally define what symbolic execution of programs constructed using the syntax in Figure 2 stands for.

For the annotated version of the `AbsValue` program, notice that the `if` statement goes through each of its branches depending on whether `x` is negative or not. When reasoning about `AbsValue`, instead of considering every possible value for `x`, we can think of in terms of a symbolic value `X` for `x`, denoted `x:X`, and then distinguish two possible cases: either

`X < 0` or `X >= 0`. For convenience, we assume there exists a symbolic value `Y` for `y`, *i.e.* `y:Y`. Now, let us try to show whether `AbsValue` is correct.

i. Case `X < 0`. If `X < 0` the guard of the `if` statement will hold and the `then` clause will be executed, hence `Y` is assigned to the negative value of `X`, *i.e.* `-X`[4]. The execution proceeds then to the `assert` annotation in which case we should be able to deduce:

```
(-X = X | -X = -X) & -X >= 0 & X = X
```

The first and last conjunctions are trivially `true`. Showing that `-X >= 0`, *i.e.* `X < 0`, follows from the case assumption.

i. Case `X >= 0`. In this case the `else` branch of the `if` statement is executed and `Y` becomes `X`. The program correctness is thus reduced to proving:

```
(X = X | X = -X) & X >= 0 & X = X
```

The first and last conjunctions reduce trivially to `true`. Again, the second conjunction follows from the case assumption.

Notice that this proof can be seen as a Hoare logic proof [9]. The added value consists in that, when doing symbolic reasoning, variables are not "attached" to a specific value.

2.3 Symbolic execution in more detail.

When executing a program symbolically, the execution goes through symbolic states. A symbolic state is a tuple $(x_i{:}X_i;pc{:}Bool)$ composed of symbolic variables X_i for variables x_i, and a so-called *path condition* `pc`. The path condition is a quantifier-free boolean formula over the symbolic inputs, it accumulates constraints which the input 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). We define the meaning of the symbolic execution of programs 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 arithmetic operator `+`, `-`, etc.

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

(*iii.*) *Symbolic execution of conditional statements.* The symbolic execution of a conditional statement `if(Cond)` Body_1 `else` Body_2 becomes more intricate than the execution described for the two previous cases; the symbolic execution of a conditional statement is resumed in the following steps. *(a.)* First, evaluate the boolean expression `Cond`[5], *(b.)* If

[4] Note that `X = pre(x)`.

[5] Note that this requires of counting on a decision procedure. We will not go further on this topic in the rest of this section; we just assume that this decision procedure exists.

the current path condition pc implies Cond, then no new sub-cases are necessary because pc already contains information enough to deduce that Body_1 must be executed. If pc implies !Cond then similarly the path condition does not require to be updated because it already contains information enough to deduce that Body_2 must be executed. Notice that the case when the evaluated boolean expression Cond reduces to false or true is covered by *(b.)*. *(c.)* Establish a sub-case where the path condition of the current symbolic state is changed from pc to pc & Cond, then proceed with the symbolic execution of Body_1. *(d.)* Establish a sub-case where the path condition of the current path condition is changed from pc to pc & !Cond, then proceed with the symbolic execution of Body_2.

(*iv.*) *Symbolic execution of annotations.* One can also define the symbolic execution of program annotations. For symbolically executing the input annotation assume Exp ; first evaluate the boolean expression Exp, *i.e.* Symb(Exp), and then update the path condition pc to pc & Symb(Exp). For output annotations assert Exp ;, first evaluate symbolically the expression Exp, *i.e.* Symb(Exp). If pc implies Symb(Exp) then the program is correct, otherwise the program fails.

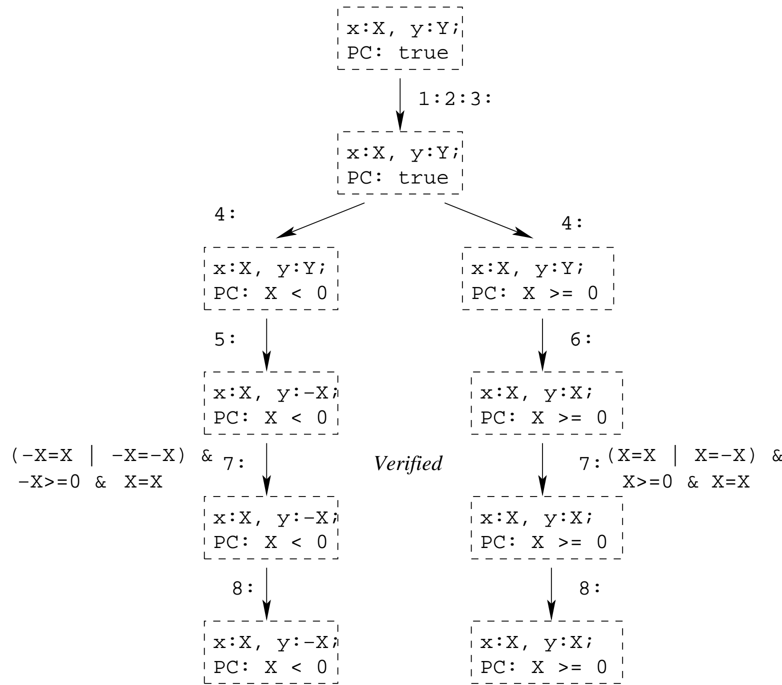


Figure 3: Symbolic execution tree for AbsValue program

Figure 3 shows the symbolic execution tree corresponding to the AbsValue annotated program. A symbolic tree has symbolic states as nodes, and additionally we have labeled

the transitions with program instructions. Two nodes $\mathtt{s}_1$ and $\mathtt{s}_2$ are connected in the graph by means of label `l:`, if the symbolic execution of the instruction `l:` (following Item (*i.*) to Item (*iv.*) above) in the symbolic state $\mathtt{s}_1$ yields the state $\mathtt{s}_2$.

Note that, unlike assignments, `if` statements have two outgoing arrows, each for each possible case resulting from assuming the guard or its negation. Unlike `if` statements, assignments update the path condition.

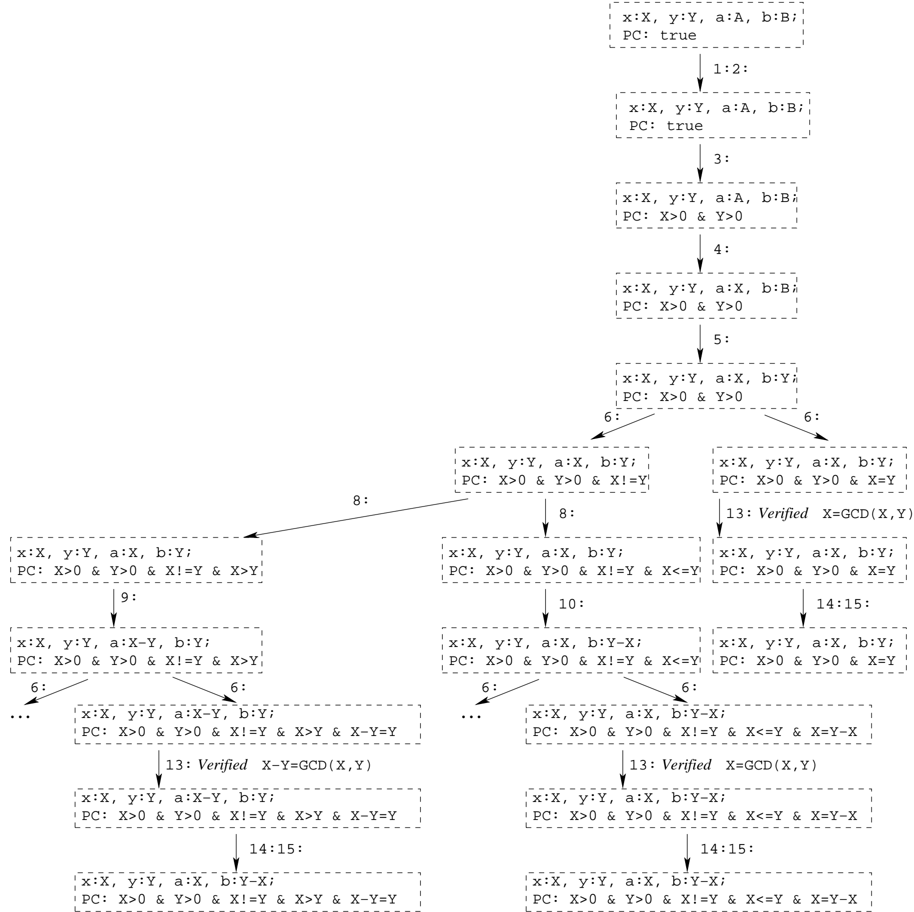


Figure 4: Symbolic execution tree for `GreatestCommonDivisor` program

In the following we go through the details about symbolic execution of cycles. The symbolic execution of a `while` statement follows naturally from the symbolical execution of an `if` statement, *i.e.* the guard of the `while` is symbolically evaluated as in (*iii.*), and further

one must consider the case when after executing the body of the `while` one goes (again) on the guard, whose execution might add a new path condition. Therefore, the symbolic execution execution of a `while` might generate an infinite tree. Notice that the presence of this infinite tree does not actually depend on whether the `while` loops forever, but on the fact of adding a new constraint path each time the guard is evaluated. Of course, an infinite loop gives rise to an infinite symbolic tree.

Below we present the code for `GreatestCommonDivisor`, which aims at calculating the *greatest common divisor* for two positive integers `x` and `y`. The code for `GreatestCommonDivisor` consists of a `while` loop, inside of which an `if` statement is executed. The output correctness annotation (Line `13:`) states that the returned value `a` is the greatest common divisor for `x` and `y`, `GCD(x,y)`. We assume that the function `GCD` knows how to calculate the greatest common divisor. Figure 4 shows the (infinite) symbolic execution tree for `GreatestCommonDivisor`. How can the method described previously be applied when dealing with infinite symbolic execution trees? That will be the topic of the next section.

```
1:  procedure GreatestCommonDivisor(x,y)
2:   declare x, y, a, b : integer;
3:   assume x > 0 & y > 0;
4:   a = x;
5:   b = y;
6:   while (a != b) do
7:
8:    if(a > b)
9:      then a := a - b;
10:     else b := b - a;
13:  assert a = GCD(x,y);
14:  return (a);
15: endp;
```

2.4 Program correctness for infinite symbolic execution trees.

The method previously introduced for proving program correctness works fine for *finite* symbolic trees. However, when loops are considered, the same reasoning cannot be applied any longer. One should make infinite symbolic trees finite.

The source of infiniteness for the symbolic tree in Figure 4 effectively is the adding of yet another new path condition each time the `while` guard (Line `6:` in `GreatestCommonDivisor` above) is visited. We should thus "cut" transversally this `while` loop in order to render the tree finite. The cut point is placed after the guard is evaluated and before any body instruction. In our example, the cutting point should be place at Line `7:`. Call the symbolic execution tree starting at the cutting point and finishing immediately after the loop body is executed *cut symbolic execution tree*. Since one wishes that the cut tree is representative of any cut symbolic execution tree, and because one does not have any previous knowledge about neither the path condition which leads the execution to the cutting point nor the

symbolic values of variables, one must consider the symbolic state at the cutting point to have the path condition `pc` initialized to `true`[6] and having symbolic values for every loop variable.

With the previous construction, a symbolic execution starting at the cutting point and reaching the end of the loop is representative of any cut symbolic tree, as we are considering the cut point symbolic state to be the most general one. The proof of correctness for the (infinite) entire symbolic tree can be constructed from proofs for the (finite) cut symbolic trees.

To do this proof, one just needs the input-output annotations for the cut symbolic trees, where the input is assumed at the beginning of the loop and the output asserted at the end. In literature, this kind of "unchanging" properties are called "invariants". They are called "inductive" because they correspond with the inductive hypothesis in the usual proof by mathematical induction.

Now, suppose that "by magic" one knows of some property `I` to be an invariant for an execution tree, our correctness proof program now looks as below. The execution starts with the general symbolic state `(x:X,y:Y,a:A,b:B;pc:true)`. In Line `7:` the invariant `I` is assumed and from Lines `8:` to `10:` the loop body is executed. Additionally, Lines `11:` and `12:` ensure that it is always possible to unroll once again the loop, and lines `11:` and `13:` that once the loop finished to do its work, the output assertion must hold. Figure 5 presents the (finite) symbolic tree for the loop body.

```
.
.
.
7:   assume I;
8:   if (a > b)
9:     then a := a - b;
10:    else b := b - a;
11:  if (a != b)
12:    then assert I;
13:    else assert a = GCD(x,y);
14:         return (a);
15: endp;
```

Something is missing in our reasoning. In fact, what we described above was the correctness proof for the case *k to k+1* in the whole by induction proof; one still needs to prove that `I` holds for the base case. In other words, one needs to prove that `I` holds regardless of the symbolic state at the initial program point. The initial program point can be seen a "by default" cutting point. Below we shows the base case for our `GreatestCommonDivisor` program. Lines `6:` and `12:` ensure that the invariant `I` is reachable from the program cutting point. Notice that we do not consider the loop guard as part of the invariant because actually the guard might become `false` at the end of the loop body. Line `13:` says that

[6]The most general boolean condition, since it is implied by any condition.

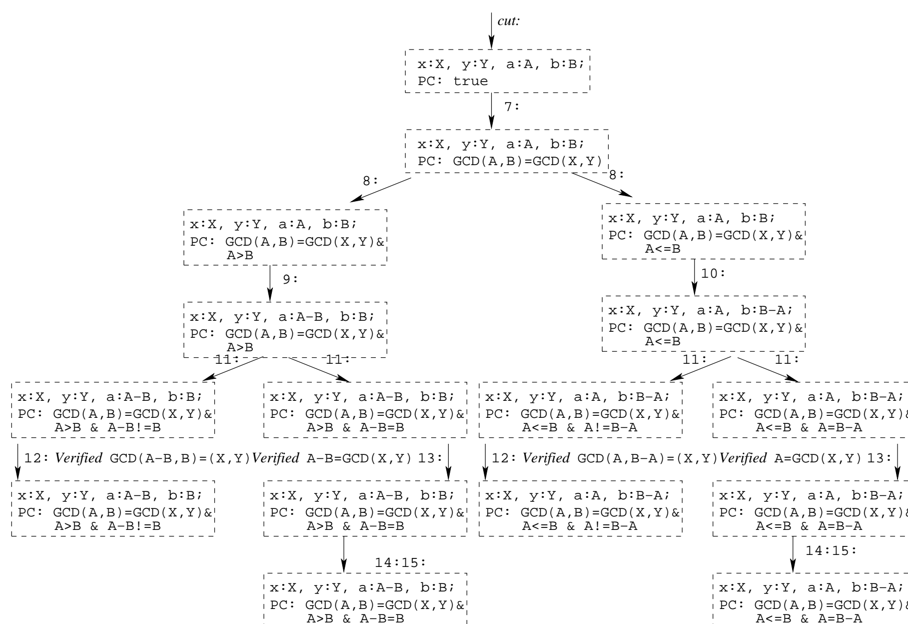


Figure 5: k-step for `GreatestCommonDivisor`

regardless of whether `I` is a correct loop invariant, the output program annotation must hold. Figure 6 shows the (finite) symbolic execution tree for the base case. Reader is invited to consult [12, 13, 6] for more information about symbolic execution.

```
1:  procedure GreatestCommonDivisor(x,y)
2:   declare x, y, a, b : integer;
3:   assume x > 0 & y > 0;
4:   a = x;
5:   b = y;
6:   if (a != b)
12:     then assert I;
13:    else assert a = GCD(x,y);
14:         return (a);
15: endp;
```

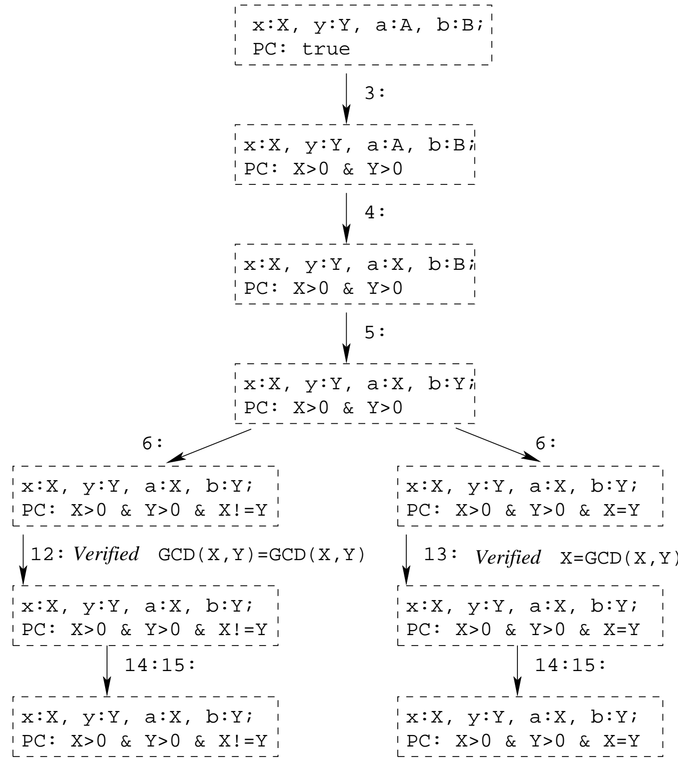


Figure 6: Base case for `GreatestCommonDivisor`

3 Symbolic execution in Java PathFinder

A symbolic execution based framework has been recently built on top of the Java PathFinder model checker (JPF) [16]. JPF is an explicit state model checker for Java programs that has been built on top of a custom-made Java Virtual Machine (JVM). It 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 model checker was 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 [15]). An annotation `ignoreIf(cond)` is used to allow backtracking — whenever `cond` evaluates to `true` the model checker will stop exploring the branch and backtrack.

Symbolic execution techniques traditionally arose in the context of sequential programs with a fixed number of integer variables. In [16], these techniques have been extended to handle dynamically allocated data structures, *e.g.*, lists and trees, complex preconditions,

e.g. disallowing cyclic lists, other primitive data, *e.g.* 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 fact allows symbolic execution of procedures without requiring an a priori bound on the number of input objects. Procedure preconditions are used to initialize inputs only with valid values.

In the following we give more specificities about the symbolic execution framework of JPF.

Algorithm. To execute a method `m` in some class `C`, the algorithm for symbolic execution in JPF first creates a new object `o` (of class `C`) whose fields are not initialized. The algorithm invokes the method `m` of object `o`, *i.e.* `o.m()`, and then the execution goes on, following the Java semantics for operations on reference fields and the symbolic execution for expressions described in Section 2. However, the access to uninitialized fields follows a special treatment. In the following we describe how the access to uninitialized fields are treated in JPF.

1. When a program accesses an uninitialized reference field, the algorithm nondeterministically initializes that field to `null`, or to a reference to a new object with uninitialized fields, or to a reference to an object created during a previous field initialization. When execution accesses an uninitialized primitive field or a string, the algorithm first initializes the field to a new symbolic value of the appropriate type, and then the execution goes on.

   Method preconditions are used to ensure that fields are initialized to values permitted by the precondition: when a reference field is initialized, the algorithm checks that the precondition does not fail for the structure and the path condition that currently constrains the object `o`.

2. If the execution evaluates a branching condition on primitive fields, the algorithm nondeterministically adds the condition or its negation to the corresponding path condition and checks whether the path condition is satisfiable. To do that, a decision procedure provided by the Omega is used [15]. If the path condition becomes `false`, the algorithm backtracks; otherwise the execution proceeds.

Example. Class `Node` below implements single-linked lists. Fields `elem` and `next` represent the value of the node and the reference to the next node. Method `swapNode` destructively updates its input list, referenced by the implicit parameter `this`, to sort its first two nodes and returns the resulting list.

```
class Node {
  int elm;
  Node next;
```

```
    Node swapNode() {
1:    if (next != null) {
2:      if (elem-(next.elem) > 0) {
3:        Node t = next;
4:        next = t.next;
5:        t.next = this;
6:        return t;
        }
      }
7:    return this;
    }
  }
```

When following the algorithm on class `Node` a new object is created on which the method `swapNode` is invoked. Line `1:` accesses the uninitialized `next` field and causes it to be initialized. Then the algorithm explores three possibilities: either the field is `null`, or the field points to a new symbolic object or it points to a previously created object of the same type. Another initialization happens during the execution of statement at Line `4:`, which results in four possibilities since there exist two objects of type `Node` at that point in the execution.

When a condition involving primitive fields is executed, *e.g.* Line `2:`, the execution tree has a branch corresponding to each possible outcome of the evaluation of the condition. The evaluation of a condition involving reference fields does not cause branching, unless uninitialized fields are accessed.

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 (in general) undecidable when states represent path conditions on unbounded data. Also, as discussed in Sections 2, performing symbolic execution on programs with loops can explore infinite execution trees. To overcome this, Section 5 describes how we try with cycles in Java PathFinder.

4 Eliminating useless emissions for Java sequential programs

We use model checking and symbolic reasoning to prove that we can remove useless emissions from the program and still *preserve* the semantics of the observer; the observer's semantics is as "the capability of reacting under the same program conditions", and formulated by the predicate `React` (see Section 1).

Consider the instrumented program below where an emission statement has been added immediately after each modification of variable `x` — the only variable in the program. Also,

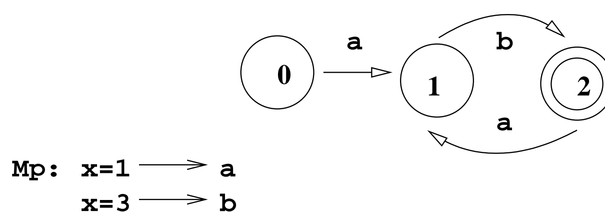


Figure 7: Mapping for `(ab)`$^+$

consider the observer for the property `(ab)`$^+$ in Figure 7, and the event mapping relation `Mp` which associates the event `x=1` in the program with the automaton event symbol `a`, and `x=3` with `b`. The automaton in Figure 7 only reacts when variable `x` takes value 1 or 3. Variable `x` is initially given a nonnegative value as returned by function `random`; after the fourth increment of `x`, that value will be greater or equal than 4. Hence, the last emission will produce no reaction in the automaton — `Stay(4,q)(a)` for every `q` $\in$ `Q` will be valid —, so that emission becomes useless and hence (should) 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);
}
```

Afterward we present how the symbolic version for the program above is first elaborated and then model checked in the symbolic execution framework of JPF. We prove that the last emission can be removed from the instrumented program, while still preserving the semantics of the observer.

The symbolic version for our program is presented below. Variable `x` of type `int` has been replaced by the variable `X` of type `SymbolicInteger` — an integer implementation for `Expression`. Classes `SymbolicInteger` and `Expression` are defined in the module for symbolic execution constructed on the top of JPF.

```
static void m() {
 0: Expression X = new SymbolicInteger();
    _addDet(GE,X,0);
    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 creating the symbolic value `X`, the condition "values for `X` are always greater or equal than 0" is added — `_addDet(GE,X,0)`: this reflects the fact that `random` in the original program always returns a nonnegative integer value. Moreover, operations increasing `x` have been replaced by method calls to `X` — `X._plus(1)`.

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

The figure 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 Java PathFinder model checker on the whole symbolic program, 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 current path condition on `X`, needed to get the program location `loc` from the starting condition `x=x_init`[7]. 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 symbolically executing the whole program 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 never produce a reaction on the automaton for any initial value of `x`. Therefore, that emission can be removed, while preserving the semantics of the observer.

Consequently, 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.

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

The 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 of x, x_init.

```
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 5 extends the work presented in the current section to include cycles.

5 Eliminating cycles' useless emissions

When considering cycles, a similar analysis as in Section 4 cannot be done: when doing symbolic execution, the Java PathFinder model checker cannot determine whether a state has been previously revised. 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 invariant for the loop, we can decide to emit only in those cases where the emission does not contradict the invariant. The main problem is that, in general, finding loop invariants is an undecidable problem. We could also work with *partial* information, *i.e.* we could execute symbolically the program for a fixed number n of iterations, conjecture a *partial invariant*, and then use this partial invariant to get rid of any emission contradicting it. Obviously, the intrinsic problem is that this partial invariant could be invalidated by a subsequent loop iteration.

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

Guessing an initial conjectured invariant. We create a symbolic loop program for which all the variables controlling the way the loop iterates are symbolic. Then, we execute

this partially symbolic program and check the path conditions generated at each loop iteration. We thus conjecture a loop invariant based on the partial information at hand and finally proceed to prove that this conjecture represents a real invariant.

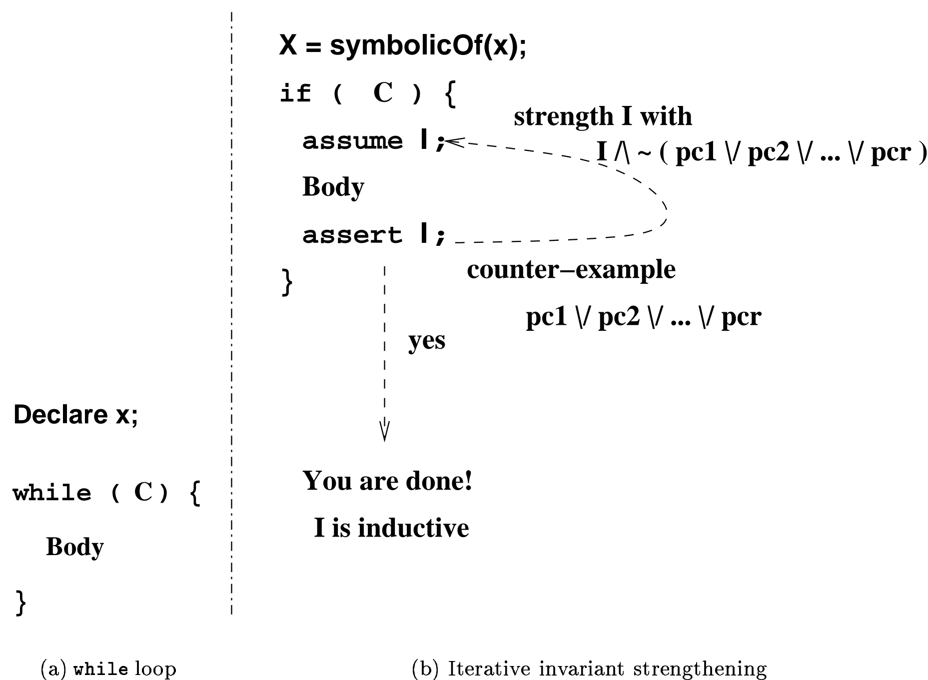
We have rewritten in a *loop style* (see below) the example presented in Section 4. The only difference resides in the fact that we now have a random number `n` of assignments to `x` instead of a fixed number 4 of iterations. Further, we have introduced variable `i` to control the current loop iteration.

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

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

Below we have a symbolic version on `X` of that loop program, where variables `i` and `n` remain concrete. We want to check the path conditions generated on the symbolic variable `X` when the symbolic program is executed for different values of `n` in order to conjecture an invariant for that loop. 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 Java PathFinder on this symbolic program, the path conditions `(x=1 || x=3)` for the first emission `i=0`, `x=0` for `i=1`, `(x=1 || x=2)` for the emission at `i=2`, and `x=0` when `i=3` are established. None of the three remaining emissions for the three remaining iterations are performed. Then, we increase the number of the iterations to `n=13`. This time, the previous conditions for location `0` and for the six first iterations of the loop remain the same; for the other `7` iterations no emission is performed. Hence, we conjecture the invariant stating that *"no emission is performed once* `i` *becomes greater than 3"* and proceed to prove whether it is an invariant.



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

Figure 8: Loop invariant strengthening

In the following we show how symbolic reasoning can be employed to prove loop invariants, and then how these invariants can be used to remove useless emissions.

Proving loop invariants by property strengthening. In general, symbolic execution of looping programs can produce *infinite* symbolic trees (see Section 2.4) because one might add yet another new path condition each time the loop guard is visited. Hence, if one wishes to use symbolic techniques to prove that an initial conjecture I_k is a loop invariant, one must first make the looping program non-looping. 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 possible condition `true`.

When the symbolic tree of the `while` program in Figure 8(a) is *cut*, we obtain the `if` program in Figure 8(b). This new program provide us with a convenient way to prove that the initially conjectured property I_k is an invariant. To prove that I_k is an invariant, I_k must be *assumed* immediately after the guard, then *asserted* immediately before finishing the body

statement. If the evaluation of that assertion succeeds, then I_k is inductive. Otherwise, if the evaluation fails, then we can employ the information provided by our symbolic program to strengthen I_k: if the execution of `assert` I_k`;` fails producing path conditions $pc_1\vee, \ldots, \vee pc_k$, then we strengthen I_k to $I_{k+1} = I_k \wedge \neg(pc_1\vee, \ldots, \vee pc_k)$ and restart the whole process, this time from I_{k+1}.

We present below the full symbolic loop program produced when using Java PathFinder for going through this iterative process of strengthening and checking. From Lines `2:` to `4:` symbolic variables for every concrete variable, included those concerned with the loop iterations, are created. From Lines `5:` to `7:`, initial conditions for these symbolic variables are added: `X` is always nonnegative, for instance. When executing symbolically an `if` statement (following Section 2.4), one should assert its guard as a condition in order to be able to execute the loop body, Line `15:`. Also, we have incorporated the initial conjecture as a necessary condition to emitting, Line `18:`. Each one of Lines `9:` to `13:` represents an iteration refinement of that initial conjecture. Each one of these refinements are again checked after the loop body has been executed, Lines `20:` to `25:`. The last iteration producing no path condition invalidating the precedent strengthened `Ik`. 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:}
```

One thing misses before this invariant is really inductive: as explained in Section 2.4, it should be valid in 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, that condition will always hold because the initial value for `i` is 0. In Line `12:`, the statement `emit(i,x)` is henceforth executed under initial conjecture condition. Only in those cases the value for `x` will be emitted. For the other cases we can not emit and still preserve the semantics of the observer, *i.e.* each time `i>3` one has `Stay(i,q)(a)` and `Stay(i,q)(b)` for any `q` in the set of states of the observer.

```
 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:}
```

5.1 Ensuring termination

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

C. Pasareanu and *W. Visser* in [14] propose an heuristic to achieve termination in this iterative process of strengthening. Yet, their problem is slightly different. They are interested in showing that a the loop program respects some property `P`[8]. They use an invariant strengthening algorithm similar to that we employ here. 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 end of the loop-body is ruled out. If there is an error in the program, the method is guaranteed to terminate. If the program is correct with respect to the property, the method might not terminate.

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

The next section describes how modular instrumentation of programs can be achieved in our framework.

6 Modularity

This section deals with the *efficient* — *i.e.* removing any emission that will not produce a reaction in the observer — instrumentation of programs when modularity is needed. We show how the instrumentation of a program can be carried out from the instrumentation of its components in isolation. But, first let us define what the instrumentation of a single statement in isolation has been in our framework following Section 4 and 5. We present our definitions using Ml syntax.

Function `instrm` below says that, when considering the automaton $\mathcal{A}$ being at the initial state q_0, the instrumentation for an assignment `x:=E` occurring inside the program `P` is calculated in the following way. First, let `loc` be the location in `P` where the assignment occurs, and `vars` be the set formed by all variables *assigned to* in `E` plus variable `x`. Then, execute symbolically the program `P` up-to location `loc` and retrieve the path condition on variables `vars` into `pc`. State q_1 is the state where the automaton $\mathcal{A}$ remains after executing program `P` up-to location `loc`.

$$
\begin{array}{l}
\texttt{instrm}\,\mathcal{A}(q_0)\ \texttt{x} := \texttt{E P} \ = \\
\quad \textbf{let}\ \texttt{loc} \ := \ \texttt{location}\,\texttt{x} := \texttt{E P} \\
\quad \textbf{and}\ \texttt{vars} \ := \ \mathcal{V}(\texttt{E}) \cup \{\texttt{x}\} \\
\quad \textbf{and}\ (\texttt{pc}, q_1) \ := \ \texttt{symbExec}\,\mathcal{A}(q_0)\ \texttt{loc vars}\ \textbf{in} \\
\qquad \texttt{x} := \texttt{E};\ \texttt{if(pc)then\{emit vars;\}}
\end{array}
$$

Consequently, the instrumentation of the assignment is given by the original assignment followed by the respective emit statement, this time under condition `pc`. Notice that assignments as expressed by `x:=E;` cover all Java statements expressed with the help of the operators `=`, `+=`, `-=`, etc; `x++` being a particular case of `x+=1`.

We express the instrumentation of other (single) statements in a similar way as for the previous function `instrm`. The instrumentation for the conditional `if(C)then{S}else{T}` is reduced to the instrumentation of `S` or `T` according to the evaluation of `C`.

Furthermore, the instrumentation for the `skip` statement — doing nothing —, can vacuously be reduced to `skip` itself when considering that the path condition `pc` evaluates to `false` whenever the set of modified variables `vars` is $\emptyset$.

Now, we are concerned with the instrumentation of the functional composition instrumentation "`;`" of statements s_1 and s_2, *i.e.* $s_1; s_2$, from the instrumentation of s_1 and s_2 in isolation. As shown below, the instrumentation of $s_1; s_2$ when considering the automaton $\mathcal{A}$ (whose initial state is q_0), is achieved by the sequential instrumentation of the first statement s_1, followed by the instrumentation of the second statement s_2 whose initial state is the final state of the symbolic execution of the first one.

$$
\begin{array}{l}
\texttt{instrm}\,\mathcal{A}\,(\texttt{q}_0)\ \texttt{s}_1;\texttt{s}_2\,\texttt{P} \quad = \\
\quad \textbf{let}\ \texttt{loc}_1 \ := \ \texttt{location}\,\texttt{s}_1\,\texttt{P} \\
\quad \textbf{and}\ \texttt{loc}_2 \ := \ \texttt{location}\,\texttt{s}_2\,\texttt{P} \\
\quad \textbf{and}\ \texttt{vars}_1 \ := \ \mathcal{V}(\texttt{s}_1) \\
\quad \textbf{and}\ \texttt{vars}_2 \ := \ \mathcal{V}(\texttt{s}_2)\ \textbf{in} \\
\quad\quad \textbf{let}\ (\texttt{pc}_1,\texttt{q}_1) \ := \ \texttt{symbExec}\,\mathcal{A}(\texttt{q}_0)\ \texttt{loc}_1\ \texttt{vars}_1 \\
\quad\quad \textbf{and}\ (\texttt{pc}_2,\texttt{q}_2) \ := \ \texttt{symbExec}\,\mathcal{A}(\texttt{q}_1)\ \texttt{loc}_2\ \texttt{vars}_2\ \textbf{in} \\
\quad\quad \texttt{s}_1;\texttt{if}(\texttt{pc}_1)\texttt{then}\{\texttt{emit}\,\texttt{vars}_1;\} \\
\quad\quad \texttt{s}_2;\texttt{if}(\texttt{pc}_2)\texttt{then}\{\texttt{emit}\,\texttt{vars}_2;\}
\end{array}
$$

The definition for functional composition instrumentation together with the previous definitions give rise to the three properties below. The two first of them say that one can "factorize" `skip` away whenever instrumented together with another statement. The last property concerns the associative property for the functional composition operator ";".

$$
\begin{array}{c}
\texttt{instrm}\,\mathcal{A}(\texttt{q}_0)\ \texttt{skip};\texttt{s}_2\,\texttt{P} = \texttt{skip};;\ \ \texttt{instrm}\,\mathcal{A}(\texttt{q}_0)\ \texttt{s}_2\,\texttt{P} \\
\texttt{instrm}\,\mathcal{A}(\texttt{q}_0)\ \texttt{s1};\texttt{skip}\,\texttt{P} = \texttt{instrm}\,\mathcal{A}(\texttt{q}_0)\ \texttt{s}_1\,\texttt{P};;\ \ \texttt{skip} \\
\texttt{instrm}\,\mathcal{A}(\texttt{q}_0)\ \text{``}\texttt{s}_1;\texttt{s}_2'';\texttt{s}_3\,\texttt{P} = \texttt{instrm}\,\mathcal{A}(\texttt{q}_0)\ \texttt{s}_1;\text{``}\texttt{s}_2;\texttt{s}_3''
\end{array}
$$

7 Conclusion

In this report 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 sampled our approach in the framework of the JPF model checker, but it 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, one needs that properties are expressed as regular expressions without self-state loops. This is not drawback of our approach, but an aspect where the JPF model checker can be improved.

Finally, the analysis presented in Section 6 explains how the process of *efficient* — *i.e.* removing as many useless emissions as possible — instrumentation can be automatized. Also, we formalize on how the instrumentation of an statement in isolation can be extended to the instrumentation achieved when putting the statement in a context where other statements are involved.

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] S.L. Hantler and J.C. King. An introduction to proving the correctness of programs. *ACM Computing Surveys*, 8(3):331–353, 1976.

[7] 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.

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

[9] C.A.R Hoare. An axiomatic basis for computer programming. In *C.A.R Hoare and C.B Jones (Ed.), Essays in Computing Science*. Prentice Hall, 1989.

[10] 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.

[11] 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.

[12] J.C. King. Proving programs to be correct. *IEEE Transactions on Computers*, 11(C-20):1331–1336, Nov. 1971.

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

[14] C. Pasareanu and W. Visser. Verification of java programs using symbolic execution and invariant generation. In *Model Checking Software: 11th International SPIN Workshop*, Barcelona, Spain, Apr. 1-3 2004.

[15] W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis, Aug. 1992.

[16] 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.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399