Concolic Execution in Functional Programming by Program Instrumentation

Adrián Palacios and Germán Vidal(✉)

MiST, DSIC, Universitat Politècnica de València,
Camino de Vera, s/n, 46022 Valencia, Spain
{apalacios,gvidal}@dsic.upv.es

Abstract. Concolic execution, a combination of concrete and symbolic execution, has become increasingly popular in recent approaches to model checking and test case generation. In general, an interpreter of the language is augmented in order to also deal with symbolic values. In this paper, in contrast, we present an alternative approach that is based on a program instrumentation. Basically, the execution of the instrumented program in a standard environment produces a sequence of events that can be used to reconstruct the associated symbolic execution.

1 Introduction

Software testing is one of the most widely used approaches for program validation. In this context, symbolic execution [8] was introduced as an alternative to random testing—which usually achieves a poor code coverage—or the complex and time-consuming design of test-cases by the programmer or software tester. In symbolic execution, one replaces the input data by symbolic values. Then, at each branching point of the execution, all feasible paths are explored and the associated contraints on symbolic values are stored. Symbolic states thus include a so called *path condition* with the constraints stored so far. Test cases are finally produced by solving the constraints in the leaves of the symbolic execution tree, which is typically incomplete since the number of states is often infinite.

Unfortunately, both the huge search space and the complexity of the constraints make test case generation based on symbolic execution difficult to scale. For instance, as soon as the path condition cannot be proved satisfiable, the execution of this branch is terminated in order to ensure soundness, giving rise to a poor coverage in many cases.

Concolic execution [4,11] is a recent proposal that combines *conc*rete and symb*olic* execution, and overcomes some of the drawbacks of previous approaches.

This work has been partially supported by the EU (FEDER) and the Spanish *Ministerio de Economía y Competitividad* under grant TIN2013-44742-C4-1-R and by the *Generalitat Valenciana* under grant PROMETEOII2015/013.

Adrián Palacios—Partially supported by the EU (FEDER) and the Spanish *Ayudas para contratos predoctorales para la formación de doctores de la Secretaría de Estado de Investigación, Desarrollo e Innovación del Ministerio de Economía y Competitividad* under FPI grant BES-2014-069749.

M. Falaschi (Ed.): LOPSTR 2015, LNCS 9527, pp. 277–292, 2015.
DOI: 10.1007/978-3-319-27436-2_17

Essentially, concolic execution takes a program and some (initially random) concrete input data, and performs both a concrete and a symbolic execution that mimics the steps of the concrete execution. In this context, symbolic execution is simpler since we know the execution path that must be followed (the same of the concrete execution). Moreover, if the path condition becomes too complex and the constraint solver cannot prove its satisfiability, we can still push some concrete data from the concrete execution, thus simplifying it and often allowing the symbolic execution to continue. This technique forms the basis of some model checking and test-case generation tools (see, e.g., SAGE [5] and Java Pathfinder [10]). Test cases produced with this technique usually achieve a better code coverage than previous approaches based solely on symbolic execution. Moreover, it scales up better to complex or large programs.

Despite its popularity in the imperative and object-oriented programming paradigms, we can only find a few preliminary approaches to concolic execution in the context of functional and logic programming. To the best of our knowledge, the first approach for a high-level declarative programming language is [13], which presented a concolic execution scheme for logic programs, which was only aimed at a simple form of *statement* coverage. This approach was later extended and improved in [9]. In the context of functional programming, [12] introduced a formalization of both concrete and symbolic execution for a simple subset of the functional and concurrent language Erlang [1], but the concolic execution procedure was barely sketched. More recently, [3] presented the design and implementation of a concolic testing tool for a complete functional subset of Erlang (i.e., the concurrency features are not considered in the paper). The tool, called CutEr, is publicly available from https://github.com/aggelgian/cuter.

However, the essential component of all these approaches is an interpreter augmented to also deal with symbolic values. In contrast, in this paper, we consider whether concolic execution can be performed by *program instrumentation*. We answer positively this question by introducing an stepwise approach based on *flattening* the initial program so that the return value of every expression is a pattern, and then instrumenting the resulting program so that its execution outputs a stream of events which suffice to reconstruct the associated symbolic execution. The main advantage w.r.t. the traditional approach to concolic execution is that the instrumented program can be run in any environment, even non-standard ones. For instance, one could run the instrumented program in a model checking environment like Concuerror [6] so that its execution would produce the sequences of events for all relevant interleavings, which might be useful for combining concolic testing and model checking.

The paper is organized as follows. Section 2 presents the considered language. Then, in Sect. 3, we present the instrumented semantics that outputs a sequence of events for each concrete execution. Section 4 introduces a program instrumentation that produces the same sequence of events but using the standard semantics. Section 5 presents a Prolog procedure for reconstructing the associated symbolic execution from the sequence of events. Finally, Sect. 6 concludes and points out some directions for further research.

$$
\begin{array}{rcl}
pgm & ::= & \mathrm{a}/n = \mathsf{fun}\ (X_1, \ldots, X_n) \to e. \mid pgm\ pgm \\
\mathsf{Exp} \ni e & ::= & \mathrm{a} \mid X \mid [\,] \mid [e_1|e_2] \mid \{e_1, \ldots, e_n\} \mid \mathsf{apply}\ e_0\ (e_1, \ldots, e_n) \\
 & & \mid \mathsf{case}\ e\ \mathsf{of}\ clauses\ \mathsf{end} \mid \mathsf{let}\ p = e_1\ \mathsf{in}\ e_2 \mid \mathsf{do}\ e_1\ e_2 \\
clauses & ::= & p_1 \to e_1; \ldots; p_n \to e_n \\
\mathsf{Pat} \ni p & ::= & [p_1|p_2] \mid [\,] \mid \{p_1, \ldots, p_n\} \mid \mathrm{a} \mid X \\
\mathsf{Value} \ni v & ::= & [v_1|v_2] \mid [\,] \mid \{v_1, \ldots, v_n\} \mid \mathrm{a}
\end{array}
$$

Fig. 1. Core erlang syntax

2 The Language

In this section, we introduce the language considered in this paper. Our language is inspired in the concurrent functional language Erlang [1], which has a number of distinguishing features, like dynamic typing, concurrency via asynchronous message passing or hot code loading, that make it especially appropriate for distributed, fault-tolerant, soft real-time applications. Erlang's popularity is growing today due to the demand for concurrent services. But this popularity will also demand the development of powerful testing and verification techniques, thus the opportunity of our research.

Despite the fact that we plan to deal with full Erlang in the future, in this paper we only consider a functional subset of *Core Erlang* [2], an intermediate language used internally by the compiler.

The basic objects of the language are variables (denoted by $X, Y, \ldots \in \mathsf{Var}$), atoms (denoted by a, b, ...) and constructors (which are fixed in Erlang to lists, tuples and atoms); defined functions are named using atoms too (we will use, e.g., f/n, g/m,...). The syntax for Core Erlang programs and expressions obeys the rules shown in Fig. 1. Programs are sequences of function definitions. Each function f/n is defined by a rule $\mathsf{fun}\ (X_1, \ldots, X_n) \to e.$ where $X_1, \ldots, X_n$ are distinct variables and the body of the function, e, can be an atom, a process identifier, a variable, a list, a tuple, a function application, a case distinction, a let expression or a do construct (i.e., $\mathsf{do}\ e_1\ e_2$ evaluates sequentially e_1 and, then, e_2, so the value of e_1 is lost). Patterns are made of lists, tuples, atoms, and variables. Values are similar to patterns but cannot contain variables.

Example 1. Consider the Erlang function (left) and its translation to Core Erlang (right) shown in Fig. 2, where some minor simplifications have been applied. Observe that Erlang's sequence operator "," is translated to a do operator when no value should be passed (using pattern matching) to the next elements in the sequence, and to a let expression otherwise. Note also that, despite the fact that this is not required by the syntax, some function applications are *flattened* in order to avoid nested applications. For this purpose, some additional let expressions are introduced. Moreover, additional default alternatives are added to each case expression in order to catch pattern matching errors, so it is common to have overlapping patterns in the clauses of a case construct.

```
f(X,Y) → g(X),                          f/2 = fun (X,Y) → do apply g/1 (X),
         case h(X) of                                     case apply h/1 (X) of
           a → A = h(Y),                                    a → let Z = apply h/1 (Y)
                g(A);                                           in apply g/1 (Z);
           b → g(h([]))                                     b → let V = apply h/1 ([])
         end.                                                   in apply g/1 (V);
                                                            W → fail
                                                         end.
```

Fig. 2. Erlang function and its translation to Core Erlang

$$
\begin{array}{rcl}
pgm & ::= & \mathrm{a}/n = \mathsf{fun}\ (X_1,\ldots,X_n) \to \mathsf{let}\ X = e\ \mathsf{in}\ X. \mid pgm\ pgm \\
\mathsf{Exp} \ni e & ::= & \mathrm{a} \mid X \mid [\,] \mid [p_1|p_2] \mid \{p_1,\ldots,p_n\} \mid \mathsf{let}\ p = e_1\ \mathsf{in}\ e_2 \mid \mathsf{do}\ e_1\ e_2 \\
 & & \mid \mathsf{let}\ p = \mathsf{apply}\ p_0\ (p_1,\ldots,p_n)\ \mathsf{in}\ e \mid \mathsf{let}\ p_1 = \mathsf{case}\ p_2\ \mathsf{of}\ clauses\ \mathsf{end}\ \mathsf{in}\ e \\
clauses & ::= & p_1 \to e_1;\ldots;p_n \to e_n \\
\mathsf{Pat} \ni p & ::= & [p_1|p_2] \mid [\,] \mid \{p_1,\ldots,p_n\} \mid \mathrm{a} \mid X \\
\mathsf{Value} \ni v & ::= & [v_1|v_2] \mid [\,] \mid \{v_1,\ldots,v_n\} \mid \mathrm{a}
\end{array}
$$

Fig. 3. Flat language syntax

As we will see later, for our instrumentation to be correct, we require some additional constraints on the syntax of programs. Basically, we require the following:

- both the name and the arguments of a function application must be patterns,
- the return value of a function must be a pattern,
- the argument of a case expression must be a pattern, and
- both function applications and case expressions can only occur in the right-hand side of a let expression.

The new constraints are needed in order to keep track of the intermediate values returned by expressions. These values are stored in a pattern, which can then be used by other expressions or returned as the result of a function application.

The restricted syntax is shown in Fig. 3. In the following, we call the programs fulfilling this syntax *flat programs*. In practice, one can transform (purely functional) Core Erlang programs to our flat syntax using a simple pre-processing transformation. Furthermore, in the flat language we also require the *bound* variables in the body of the functions to have unique, fresh names. This is not strictly necessary, but it simplifies the presentation by avoiding the use of context scopes associated to every let expression, etc. (as in [7], where the *last* binding of a variable in the environment should be considered to ensure that the right scope is used). We denote with $\overline{o_n}$ a sequence of objects $o_1,\ldots,o_n$. $\mathcal{V}ar(e)$ denotes the set of variables appearing in an expression e, and we say that e is *ground* if $\mathcal{V}ar(e) = \emptyset$.

In the following, we use the function bv to gather the bound variables of an expression:

Definition 1 (Bound Variables, bv). *Let e be an expression. The function* $\mathsf{bv}(e)$ *returns the set of bound variables of e as follows:*

$$\mathsf{bv}(e) = \begin{cases} \{\,\} & \text{if } e \in \mathsf{Pat} \\ \mathcal{V}ar(p) \cup \mathsf{bv}(e') & \text{if } e \equiv \mathsf{let}\ p\ = \mathsf{apply}\ p_0\ (p_1, \ldots, p_n)\ \mathsf{in}\ e' \\ \mathcal{V}ar(p_0) \cup \ldots \cup \mathcal{V}ar(p_n) \cup \mathsf{bv}(e_1) \cup \ldots \cup \mathsf{bv}(e') & \text{if } e \equiv \mathsf{let}\ p_0 = \mathsf{case}\ p\ \mathsf{of}\ \overline{p_n \to e_n}\ \mathsf{end}\ \mathsf{in}\ e' \\ \mathcal{V}ar(p) \cup \mathsf{bv}(e_1) \cup \mathsf{bv}(e_2) & \text{if } e \equiv \mathsf{let}\ p = e_1\ \mathsf{in}\ e_2 \\ \mathsf{bv}(e_1) \cup \mathsf{bv}(e_2) & \text{if } e \equiv \mathsf{do}\ e_1\ e_2 \end{cases}$$

where, in the fourth case, we assume that e_1 is neither an application nor a case expression (i.e., it is a pattern or another let expression).

3 Instrumented Semantics

In this section, we present an instrumented semantics for flat programs that produces a sequence of events that will suffice to reconstruct the associated symbolic execution. Essentially, we need to keep track of function calls, returns, let bindings and case selections.

First, let us note that the produced events will not show the actual run time values of the program variables, since they will not help us to reconstruct the associated symbolic execution. Rather, the events always include the static variable names. Therefore, in order to avoid variable name clashes, we will consider that variable names are *local* to every event. As a consequence, the two first elements of all events are *params* and *vars* denoting the list of parameters and the list of bound variables in the current function, respectively. These elements will be matched with the current values in the symbolic execution built so far in order to set the right environment for the operation represented by the event. See Sect. 5 for more details.

We consider the following events, which will suffice to reconstruct the symbolic execution:

- The first event, $\mathsf{call}(params,vars,p,[p_1,\ldots,p_n])$, is associated to a function application $\mathsf{let}\ p = \mathsf{apply}\ p_0\ (p_1,\ldots,p_n)\ \mathsf{in}\ e$. Here, $[p_1,\ldots,p_n]$ are the arguments of the function call, and p will be used to store the *return value* of the function call.
- The second event is $\mathsf{exit}(params,vars,p)$, where p is the pattern used to store the return value of the function body. We will produce an *exit* event at the end of every function.
- The next event is $\mathsf{bind}(params, vars, p, p')$, which binds the pattern p from a generic let expression (i.e., a let expression whose argument is neither an application nor a case expression) to the return value p' of that expression (see function ret below).
- Finally, for each expression of the form

$$\mathsf{let}\ p = \mathsf{case}\ p_0\ \mathsf{of}\ p_1 \to e_1; \ldots; p_n \to e_n\ \mathsf{end}\ \mathsf{in}\ e$$

we have two associated events. The first one is

$$\mathsf{case}(params, vars, i, p_0, p_i, [(1, p_0, p_1), \ldots, (n, p_0, p_n)])$$

Here, we store the position of the selected branch, i, the case argument p_0, the selected pattern p_i, as well a list with all case branches, which will become useful for producing alternative input data in the context of concolic testing. The second event is $\mathsf{exitcase}(params,vars,p,p')$, where p' is the return value of the selected branch (see below).

Before presenting the instrumented semantics, we need the following auxiliary function that identifies the *return value* of an expression:

Definition 2 (Return Value, ret). *Let e be an expression. We let $\mathsf{ret}(e)$ denote the return value of e as follows:*

$$\mathsf{ret}(e) = \begin{cases} e & \text{if } e \in \mathsf{Pat} \\ \mathsf{ret}(e') & \text{if } e \equiv \mathsf{let}\ p\ = \mathsf{apply}\ p_0\ (p_1, \ldots, p_n)\ \mathsf{in}\ e' \\ \mathsf{ret}(e') & \text{if } e \equiv \mathsf{let}\ p_0 = \mathsf{case}\ p\ \mathsf{of}\ \overline{p_n \to e_n}\ \mathsf{end}\ \mathsf{in}\ e' \\ \mathsf{ret}(e_2) & \text{if } e \equiv \mathsf{let}\ p = e_1\ \mathsf{in}\ e_2 \\ \mathsf{ret}(e_2) & \text{if } e \equiv \mathsf{do}\ e_1\ e_2 \end{cases}$$

where, in the fourth case, we assume that e_1 is neither an application nor a case expression (i.e., it is a pattern or another let expression).

Note that function ret is not well defined for arbitrary programs, e.g., $\mathsf{ret}(\mathsf{let}\ p\ = e\ \mathsf{in}\ \mathsf{apply}\ e_0\ (e_1, \ldots, e_n))$ is undefined. Extending the definition to cover this case would not help too since returning an expression which is not a pattern—like $\mathsf{apply}\ e_0\ (e_1, \ldots, e_n)$—would not be useful to reconstruct the symbolic execution (where the program is not available, only the sequence of events). This is why we transform the original programs to the flat form. In this case, it is immediate to see from the syntax in Fig. 3 that ret would always return a pattern for all program expressions.

The instrumented semantics for flat programs is formalized in Fig. 4 following the style of a natural (big-step) semantics [7]. Observe that we do not need *closures* (as it is common in the natural semantics) since we do not allow fun expressions in the body of a function in this paper. Here, we use an *environment θ*—i.e., a mapping from variables to patterns—because we need to know the static values of the variables for the instrumentation (e.g., we use the case argument that appears statically in the program, rather than the instantiated run time value). The main novelty is that, for the instrumentation, we also need to keep track of the function where an expression occurs. For this purpose, we also introduce a simple context π that stores this information, i.e., for a given function $\mathsf{fun}\ (X_1, \ldots, X_n) \to e$ we store a tuple $\langle [X_1, \ldots, X_n], [\mathsf{bv}(e)] \rangle$. The environment is only updated in function applications, where $[\mathsf{bv}(e)]$ denotes a list with the variables returned by $\mathsf{bv}(e)$.

Let us briefly explain the rules of the semantics. Statements have the form $\pi, \theta \vdash e \Downarrow_\tau p$, where π is the aforementioned context, θ is a substitution (the

$$\frac{}{\pi, \theta \vdash p \Downarrow_{\epsilon} p\theta}$$

$$\frac{\langle vs, ps\rangle, \theta \vdash p_0 \Downarrow_{\epsilon} \mathrm{f}/m \quad \ldots \quad \langle vs, ps\rangle, \theta \vdash p_m \Downarrow_{\epsilon} p'_m \quad \langle [\overline{Y_m}], [\mathsf{bv}(e_2)]\rangle, \theta \cup \sigma \vdash e_2 \Downarrow_{\tau_1} p' \quad \langle vs, ps\rangle, \theta \cup \sigma' \vdash e \Downarrow_{\tau_2} p''}{\langle vs, ps\rangle, \theta \vdash \mathsf{let}\ p = \mathsf{apply}\ p_0\ (\overline{p_m})\ \mathsf{in}\ e \Downarrow_{\mathsf{call}(vs,ps,p,[\overline{p_m}])+\tau_1+\mathsf{exit}([\overline{Y_m}],[\mathsf{bv}(e_2)],p''_2)+\tau_2} p''}$$

$$\text{if } \mathrm{f}/m = \mathsf{fun}\ (\overline{Y_m}) \to e_2 \in \mathsf{pgm},\ \mathsf{ret}(e_2) = p''_2,\ \mathsf{match}(\overline{Y_m}, \overline{p'_m}) = \sigma,\ \mathsf{match}(p, p') = \sigma'$$

$$\frac{\langle vs, ps\rangle, \theta \vdash p_0 \Downarrow_{\epsilon} p'_0 \quad \langle vs, ps\rangle, \theta \cup \sigma \vdash e_i \Downarrow_{\tau_1} p'_i \quad \langle vs, ps\rangle, \theta \cup \sigma' \vdash e \Downarrow_{\tau_2} p'}{\langle vs, ps\rangle, \theta \vdash \mathsf{let}\ p = \mathsf{case}\ p_0\ \mathsf{of}\ \mathit{clauses}\ \mathsf{end}\ \mathsf{in}\ e \Downarrow_{\mathsf{case}(vs,ps,i,p_0,p_i,alts)+\tau_1+\mathsf{exitcase}(vs,ps,p,p'_i)+\tau_2} p'}$$

$$\text{if } \mathit{clauses} = p_1 \to e_1; \ldots; p_m \to e_m,\ \mathsf{cmatch}(p'_0, \mathit{clauses}) = (i, p_i, \sigma),\ \mathit{alts} = [(1, p_0, p_1), \ldots, (m, p_0, p_m)],\ \mathsf{ret}(e_i) = p'_i,\ \mathsf{match}(p, p'_i) = \sigma'$$

$$\frac{\pi, \theta \vdash e_1 \Downarrow_{\tau_1} p'_1 \quad \pi, \theta \cup \sigma \vdash e_2 \Downarrow_{\tau_2} p}{\pi, \theta \vdash \mathsf{let}\ p_1 = e_1\ \mathsf{in}\ e_2 \Downarrow_{\tau_1+\mathsf{bind}(vs,ps,p_1,\mathsf{ret}(e_1))+\tau_2} p} \quad \text{if } \mathsf{match}(p_1, p'_1) = \sigma$$

$$\frac{\pi, \theta \vdash e_1 \Downarrow_{\tau_1} p_1 \quad \pi, \theta \vdash e_2 \Downarrow_{\tau_2} p_2}{\pi, \theta \vdash \mathsf{do}\ e_1\ e_2 \Downarrow_{\tau_1+\tau_2} p_2}$$

Fig. 4. Flat language instrumented semantics

environment), e is an expression, τ is a sequence of events, and p is a pattern—the value of e.

The first rule deals with patterns (including variables, atoms, tuples and lists). Here, the evaluation just proceeds by applying the current environment θ to the pattern p to bind its variables (if any), which is denoted by $p\theta$. The associated sequence of events is ϵ denoting an empty sequence.

The next rule deals with function applications. In this case, the context is necessary for setting the first and second parameters of call and exit events. Note that since we only consider flat programs, both the function name and the arguments are patterns; thus, their evaluation amounts to binding their variables using the current environment, which explains why the associated sequences of events are ϵ. Note also that, when recursively evaluating the body of the function, we update the context with the information of the function called. The bound variables are collected using the function bv; and, as mentioned before, in the flat language we assume that they all have different, fresh names. Observe that the subcomputation for evaluating the body of the function called is preceded by the call event and followed by an exit event. Here, we use the auxiliary function match to compute the matching substitution (if any) between two patterns, i.e., $\mathsf{match}(p_1, p_2) = \sigma$ if $\mathcal{D}om(\sigma) \subseteq \mathcal{V}ar(p_1)$ and $p_1\sigma = p_2$, and fail otherwise. In this rule, $\mathsf{match}(\overline{Y_m}, \overline{p'_m})$ just returns the substitution $\{Y_1 \mapsto p'_1, \ldots, Y_m \mapsto p'_m\}$. The *update* of an environment θ using σ is denoted by $\theta \cup \sigma$. Formally, $\theta \cup \sigma = \delta$ such that $X\delta = \sigma(X)$ if $X \in \mathcal{D}om(\sigma)$ and $X\delta = X\theta$ otherwise (i.e., σ has higher priority than θ). Observe that we use the evaluated patterns $p'_1, \ldots, p'_m$ to update the environment, but the original, static patterns $p_1, \ldots, p_m$ in the call event.

The next rule is used to evaluate case expressions. Here, we produce case and exitcase events that also include the parameter variables of the function and the bound variables. For selecting the matching branch of the case expression, we use the auxiliary function cmatch that is defined as follows: $\mathsf{cmatch}(p, p_1 \to e_1; \ldots; p_n \to e_n) = (i, p_i, \sigma)$ if $\mathsf{match}(p, p_i) = \sigma$ for some $i \in \{1, \ldots, n\}$ and $\mathsf{match}(p, p_j) = \mathsf{fail}$ for all $j < i$. Informally speaking, cmatch selects the first matching branch of the case expression, which follows the usual semantics of Erlang. As in the previous rule, note that we use p'_0 in cmatch but the original, static pattern p_0 in the case event.

The following rule is used to evaluate let expressions. It produces a single bind event which includes, as usual, the parameter variables of the function and the bound variables. Finally, the last rule deals with do expressions. Here, we proceed as expected and return the concatenation of the sequences of events produced when evaluating the subexpressions.

In the following, without loss of generality, we assume that the entry point to the program is always the distinguished function main/n.

Definition 3 (Instrumented Execution). *Given a flat program pgm and an initial expression,* apply main/n $(p_1, \ldots, p_n)$*, with* main/n = fun $(X_1, \ldots, X_n) \to e \in$ pgm*, its evaluation is denoted by* $\langle [\overline{X_n}], [\mathsf{bv}(e)] \rangle, \theta \vdash e \Downarrow_\tau v$*, where* $\theta = \{X_1 \mapsto p_1, \ldots, X_n \mapsto p_n\}$ *is a substitution,* v *is the computed value and* $\tau +$ $\mathsf{exit}([\overline{X_n}], [\mathsf{bv}(e)], \mathsf{ret}(e))$ *is the associated sequence of events.*

Example 2. Let us consider the flat program shown in Fig. 5. An example computation for apply main/1 ([a]) with the instrumented semantics is shown in Fig. 6. Therefore, the associated sequence of events[1] is the following:

$$\begin{array}{l}
\mathsf{call}([X], [W], W, [X, X]) \\
\mathsf{case}([X, Y], [W_1, H, T, W_2], 2, X, [H|T], [(1, X, [\,]), (2, X, [H|T])]) \\
\mathsf{call}([X, Y], [W_1, H, T, W_2], W_2, [T, Y]) \\
\mathsf{case}([X, Y], [W_1, H, T, W_2], 1, X, [\,], [(1, X, [\,]), (2, X, [H|T])]) \\
\mathsf{exitcase}([X, Y], [W_1, H, T, W_2], W_1, Y) \\
\mathsf{exit}([X, Y], [W_1, H, T, W_2], W_1) \\
\mathsf{exitcase}([X, Y], [W_1, H, T, W_2], W_1, [H|W_2]) \\
\mathsf{exit}([X, Y], [W_1, H, T, W_2], W_1) \\
\mathsf{exit}([X], [W], W)
\end{array}$$

Let us remind that variable names are *local* to each event. Also, observe that the events do not need to store the names of the invoked functions since we are only interested in the sequence of pattern matching operations, as we will see in Sect. 5.

Note that the semantics is a conservative extension of the standard semantics in the sense that the generation of events does not affect the evaluation, i.e., if we

[1] Note that the flat program is not syntactically correct according to Fig. 3 since the right-hand side of the functions do not have the form let $X = e$ in X with e a pattern, a let binding or a do expression. Here, we keep this simpler formulation for clarity, and it also simplifies the sequence of events by avoiding some redundant bind events.

$$
\begin{array}{l}
\mathrm{main}/1 \;=\; \mathsf{fun}\ (X) \to \mathsf{let}\ W = \mathsf{apply}\ \mathrm{app}/2\ (X, X)\ \mathsf{in}\ W \\
\mathrm{app}/2 \;=\; \mathsf{fun}\ (X, Y) \to \mathsf{let}\ W_1 = \mathsf{case}\ X\ \mathsf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad [\,] \to Y \\
\qquad\qquad\qquad\qquad\qquad\qquad [H|T] \to \mathsf{let}\ W_2 = \mathsf{apply}\ \mathrm{app}/2\ (T, Y)\ \mathsf{in}\ [H|W_2] \\
\qquad\qquad\qquad\qquad\qquad \mathsf{end} \\
\qquad\qquad\qquad\qquad \mathsf{in}\ W_1
\end{array}
$$

Fig. 5. Example flat program

remove the context information and the events labeling the arrows, we are back to the standard semantics of an eager functional language essentially equivalent to that in [7].

We will show a method for constructing the associated symbolic execution (as well as its potential alternatives) in Sect. 5.

4 Program Instrumentation

In this section, we present a program transformation that instruments a program so that its standard execution will return the same sequence of events produced with the original program and the instrumented semantics of Fig. 4.

For this purpose, we introduce the predefined function out, which outputs its first argument (e.g., to a given file or to the standard output) and returns its second argument. This function is implemented as a function *call* (i.e., not as a function application) so that there is no conflict when performing the instrumentation.

Definition 4 (Program Instrumentation). *Let pgm be a flat program. We instrument pgm by replacing each function definition:*

$$\mathrm{f}/k = \mathsf{fun}\ (X_1, \ldots, X_k) \to \mathsf{let}\ X = e\ \mathsf{in}\ X$$

with a new function definition of the form

$$\mathrm{f}/k = \mathsf{fun}\ (X_1, \ldots, X_k) \to [\![\mathsf{let}\ X = e\ \mathsf{in}\ \mathsf{out}(\text{“}\mathsf{exit}(vs, bs, X)\text{”}, X)]\!]_{\mathsf{F}}^{vs,ps}$$

where $vs = [\overline{X_k}]$, $ps = [\mathsf{bv}(e)]$, F *is a flag to determine if an* exitcase *event should be produced when a pattern is reached (see below), and the auxiliary function* $[\![\ ]\!]$ *is shown in Fig. 7.*

Let us briefly explain the rules of the instrumentation. First, we add an exit event at the end of each function. An additional bind event is also required when the expression e is neither a function application nor an case expression in order to explicitly bind X to the return expression of e (for function applications and case expressions this is already done in the exit and exitcase events, respectively).

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\pi_2,\theta_4 \vdash Y \Downarrow_\epsilon [\mathrm{a}] \qquad \pi_2,\theta_5 \vdash W_1 \Downarrow_\epsilon [\mathrm{a}]}{\pi_2,\theta_4 \vdash \mathsf{let}\ W_1 = \mathsf{case}\ldots \Downarrow_{\tau_1} [\mathrm{a}]} \qquad \pi_2,\theta_6 \vdash [H|W_2] \Downarrow_\epsilon [\mathrm{a},\mathrm{a}]}{\pi_2,\theta_3 \vdash \mathsf{let}\ W_2 = \mathsf{apply}\ldots \Downarrow_{\tau_2} [\mathrm{a},\mathrm{a}] \qquad \pi_2,\theta_7 \vdash W_1 \Downarrow_\epsilon [\mathrm{a},\mathrm{a}]}}{\pi_2,\theta_2 \vdash \mathsf{let}\ W_1 = \mathsf{case}\ldots \Downarrow_{\tau_3} [\mathrm{a},\mathrm{a}] \qquad \pi_1,\theta_8 \vdash W \Downarrow_\epsilon [\mathrm{a},\mathrm{a}]}}{\pi_1,\theta_1 \vdash \mathsf{let}\ W = \mathsf{apply}\ \mathrm{app}/2\ (X,X)\ \mathsf{in}\ W \Downarrow_{\tau_4} [\mathrm{a},\mathrm{a}]}$$

with

$$\pi_1 = \langle [X],[W]\rangle \quad \text{and} \quad \pi_2 = \langle [X,Y],[W_1,W_2H,T]\rangle$$

$$\begin{array}{ll}
\theta_1 = \{X \mapsto [\mathrm{a}]\} & \theta_2 = \{X \mapsto [\mathrm{a}], Y \mapsto [\mathrm{a}]\} \\
\theta_3 = \{X \mapsto [\mathrm{a}], Y \mapsto [\mathrm{a}], H \mapsto \mathrm{a}, T \mapsto [\,]\} & \theta_4 = \{X \mapsto [\,], Y \mapsto [\mathrm{a}]\} \\
\theta_5 = \{X \mapsto [\,], Y \mapsto [\mathrm{a}], W_1 \mapsto [\mathrm{a}]\} & \theta_6 = \{X \mapsto [\mathrm{a}], Y \mapsto [\mathrm{a}], H \mapsto \mathrm{a}, T \mapsto [\,], W_2 \mapsto [\mathrm{a}]\} \\
\theta_7 = \{X \mapsto [\mathrm{a}], Y \mapsto [\mathrm{a}], W_1 \mapsto [\mathrm{a},\mathrm{a}]\} & \theta_8 = \{X \mapsto [\mathrm{a}], W \mapsto [\mathrm{a},\mathrm{a}]\}
\end{array}$$

$$\begin{array}{l}
\tau_1 = \mathsf{case}([X,Y],[W_1,W_2],1,X,[\,],[(1,X,[\,]),(2,X,[H|T])]) \\
\qquad +\mathsf{exitcase}([X,Y],[W_1,W_2],W_1,Y) \\
\tau_2 = \mathsf{call}([X,Y],[W_1,W_2],W_2,[T,Y]) + \tau_1 + \mathsf{exit}([X,Y],[W_1,W_2],W_1) \\
\tau_3 = \mathsf{case}([X,Y],[W_1,W_2],2,X,[H|T],[(1,X,[\,]),(2,X,[H|T])]) + \tau_2 \\
\qquad +\mathsf{exitcase}([X,Y],[W_1,W_2],W_1,[H|W_2]) \\
\tau_4 = \mathsf{call}([X],[W],W,[X,X]) + \tau_3 + \mathsf{exit}([X,Y],[W_1,W_2],W_1)
\end{array}$$

Fig. 6. Example computation with the instrumented semantics

$$\begin{array}{rl}
[\![e]\!]^{vs,ps}_{\mathsf{F}} & = e \ \text{ if } e \in \mathsf{Pat} \\
[\![e]\!]^{vs,ps}_{\mathsf{T}(p)} & = \mathsf{out}(\text{“}\mathsf{exitcase}(vs,ps,p,e)\text{”}, e) \ \text{ if } e \in \mathsf{Pat} \\
[\![\mathsf{let}\ p = \mathsf{apply}\ p_0\ (\overline{p_n})\ \mathsf{in}\ e]\!]^{vs,ps}_b & = \mathsf{let}\ p = \mathsf{out}(\text{“}\mathsf{call}(vs,ps,p,[p_1,\ldots,p_n])\text{”}, \\
 & \qquad\qquad \mathsf{apply}\ p/0\ (p_1,\ldots,p_n)\ ) \\
 & \quad \mathsf{in}\ [\![e]\!]^{vs,ps}_b \\
[\![\mathsf{let}\ p = \mathsf{case}\ p_0\ \mathsf{of} & = \mathsf{let}\ p = \mathsf{case}\ p_0\ \mathsf{of} \\
\quad p_1 \to e_1; & \quad p_1 \to \mathsf{out}(\text{“}\mathsf{case}(vs,ps,1,p_0,p_1,alts)\text{”}, \\
 & \qquad\qquad [\![e_1]\!]^{vs,ps}_{\mathsf{T}(p)}\ ) \\
\quad \ldots & \quad \ldots \\
\quad p_n \to e_n & \quad p_n \to \mathsf{out}(\text{“}\mathsf{case}(vs,ps,n,p_0,p_n,alts)\text{”}, \\
 & \qquad\qquad [\![e_n]\!]^{vs,ps}_{\mathsf{T}(p)}\ ) \\
\quad \mathsf{end} & \quad \mathsf{end} \\
\quad \mathsf{in}\ [\![e]\!]^{vs,ps}_b & \quad \mathsf{in}\ [\![e]\!]^{vs,ps}_b \\
[\![\mathsf{let}\ p = e_1\ \mathsf{in}\ e_2]\!]^{vs,ps}_b & = \mathsf{let}\ p = [\![e_1]\!]^{vs,ps}_{\mathsf{F}}\ \mathsf{in}\ \mathsf{out}(\text{“}\mathsf{bind}(vs,ps,p,\mathsf{ret}(e_1))\text{”}, \\
 & \qquad\qquad [\![e_2]\!]^{vs,ps}_b\ ) \\
[\![\mathsf{do}\ e_1\ e_2]\!]^{vs,ps}_b & = \mathsf{do}\ [\![e_1]\!]^{vs,ps}_{\mathsf{F}}\ [\![e_2]\!]^{vs,ps}_b \\
[\![e]\!]^{vs,ps}_b & = e \ \text{ otherwise}
\end{array}$$

where $alts = [(p_0,1,p_1),\ldots,(p_0,n,p_n)]$

Fig. 7. Program instrumentation

Then, we also add call and case events in each occurrence of a function application and a case expression, respectively. Finding the value returned by a case expression is a bit more subtle. For this purpose, we introduce a flag that is propagated through the different cases so that only when the expression is the last expression in a case branch (a pattern) we produce an exitcase event. For let expressions, we produce a bind event and continue evaluating both the expression in the right-hand side of the binding and the result. Finally, the *default* case—the last equation in Fig. 7—is only used to ignore the call to the predefined function out/2.

Example 3. Consider again the flat program of Example 2. The instrumented program is shown in Fig. 8.

$$
\begin{array}{l}
\mathrm{main}/2 \;=\; \mathsf{fun}\ (X) \to \mathsf{let}\ W = \mathsf{out}(\text{“}\mathsf{call}([X],[W],W,[X,X])\text{”}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{apply}\ \mathrm{app}/2\ (X,X)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{in}\ \mathsf{out}(\text{“}\mathsf{exit}([X],[W],W)\text{”},W) \\
\\
\mathrm{app}/2 \;=\; \mathsf{fun}\ (X,Y) \to \\
\quad \mathsf{let}\ W_1 = \mathsf{case}\ X\ \mathsf{of} \\
\qquad\qquad [\,] \to\ \mathsf{out}(\text{“}\mathsf{case}([X,Y],[W_1,W_2,H,T],1,X,[\,],alts)\text{”}, \\
\qquad\qquad\qquad\qquad\qquad \mathsf{out}(\text{“}\mathsf{exitcase}([X,Y],[W_1,W_2,H,T],W_1,Y)\text{”},Y)) \\
\qquad\qquad [H|T] \to\ \mathsf{out}(\text{“}\mathsf{case}([X,Y],[W_1,W_2,H,T],2,X,[H|T],alts)\text{”}, \\
\qquad\qquad\qquad\qquad\qquad \mathsf{let}\ W_2 = \mathsf{out}(\text{“}\mathsf{call}([X,Y],[W_1,W_2,H,T],W_2,[T,Y])\text{”}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{apply}\ \mathrm{app}/2\ (T,Y))) \\
\qquad\qquad\qquad\qquad\qquad \mathsf{in}\ \mathsf{out}(\text{“}\mathsf{exitcase}([X,Y],[W_1,W_2,H,T],W_1,[H|W_2])\text{”}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad [H|W_2]) \\
\quad \mathsf{in}\ \mathsf{out}(\text{“}\mathsf{exit}([X,Y],[W_1,W_2,H,T],W_1)\text{”},W_1)
\end{array}
$$

where $alts = [(1,X,[]),(2,X,[H|T])]$.

Fig. 8. Instrumented program

It can easily be shown that the instrumented program produces the same sequence of events of Example 2, e.g., by executing the program in the standard environment of Erlang (together with an appropriate definition of out/2).

The correctness of the program instrumentation is stated in the next result:

Theorem 1. *Let pgm be a flat program and* pgm^I *its instrumented version according to Definition 4. Given an initial expression,* apply main/n $(p_1,\ldots,p_n)$*, its execution using pgm and the instrumented semantics (according to Definition 3) produces the same sequence of events as its execution using* pgm^I *and the standard semantics.*

Proof. We prove that for all program expressions, e, we have that $\langle vs, ps\rangle, \theta \vdash e \Downarrow_\tau p$ implies $\theta \vdash [\![e]\!]_{\mathsf{F}}^{vs,ps} \Downarrow p$ with the standard semantics[2] and, moreover,

[2] Here, we consider that the *standard* semantics is that of Fig. 4 without the events labeling the transitions.

it outputs the same sequence of events τ. The claim of the theorem is an easy consequence of this property. We prove the claim by induction on the depth k of the proof tree with the instrumented semantics.

Since the base case $k = 0$ is trivial (the rule to evaluate a pattern is the same in both cases), we now consider the inductive case $k > 0$. We distinguish the following cases depending on the applied rule from the semantics of Fig. 4:

- The first rule of the semantics is not applicable since the depth of the proof is $k > 0$.
- If the applied rule is the second one (to evaluate a function call), then the considered transition has the form

$$\langle vs, ps\rangle, \theta \vdash \mathsf{let}\ p = \mathsf{apply}\ p_0\ (\overline{p_m})\ \mathsf{in}\ e \Downarrow_\tau p''$$

with $\tau = \mathsf{call}(vs, ps, p, [\overline{p_m}]) + \tau_1 + \mathsf{exit}([\overline{Y_m}], [\mathsf{bv}(e_2)], p_2'') + \tau_2$. The instrumented expression is thus

$$[\![\mathsf{let}\ p = \mathsf{apply}\ p_0\ (\overline{p_m})\ \mathsf{in}\ e]\!]_b^{vs,ps}$$

Following the rules of Fig. 7, this is transformed to

$$\mathsf{let}\ p = \mathsf{out}(\text{“}\mathsf{call}(vs, ps, p, [\overline{p_m}])\text{”}, \mathsf{apply}\ p_0\ (\overline{p_m}))\ \mathsf{in}\ [\![e]\!]_b^{vs,ps}$$

such that the execution of this instrumented code will first output the event $\mathsf{call}(vs, ps, p, [\overline{p_m}])$ similarly to the instrumented semantics. By the induction hypothesis, the evaluation of $p_0, \ldots, p_m$ and e with the instrumented semantics produces the same values and outputs the same events than with their instrumented versions with the standard semantics. Let us now consider that p_0 evaluates to function f/m, whose definition is as follows: $\mathrm{f}/m = \mathsf{fun}\ \overline{Y_m} \to \mathsf{let}\ X = e\ \mathsf{in}\ X$. In the instrumented program, the same function has the form

$$\mathrm{f}/m = \mathsf{fun}\ (X_1, \ldots, X_m) \to [\![\mathsf{let}\ X = e\ \mathsf{in}\ \mathsf{out}(\text{“}\mathsf{exit}(vs, bs, X)\text{”}, X)]\!]_{\mathsf{F}}^{vs',ps'}$$

$vs' = [\overline{Y_m}]$ and $ps' = [\mathsf{bv}(e)]$. By the induction hypothesis, we know that the sequence of events for $\mathsf{let}\ X = e\ \mathsf{in}\ X$ in the instrumented semantics, is the same as that of $[\![\ \mathsf{let}\ X = e\ \mathsf{in}\ X]\!]_{\mathsf{F}}^{vs',ps'}$, therefore the claim follows.
- If the applied rule is the second one (to evaluate a function call), then the considered transition has the form

$$\langle vs, ps\rangle, \theta \vdash \mathsf{let}\ p = \mathsf{case}\ p_0\ \mathsf{of}\ \mathit{clauses}\ \mathsf{end}\ \mathsf{in}\ e \Downarrow_\tau p_0'$$

with $\mathit{clauses} = \overline{p_l \to e_l}$ and

$$\tau = \mathsf{case}(vs, ps, i, p_0, p_i, \mathit{alts}) + \tau_1 + \mathsf{exitcase}(vs, ps, p, p_i') + \tau_2$$

The instrumented expression is thus

$$[\![\mathsf{let}\ p = \mathsf{case}\ p_0\ \mathsf{of}\ \mathit{clauses}\ \mathsf{end}\ \mathsf{in}\ e]\!]_b^{vs,ps}$$

which is transformed to

$$\mathsf{let}\ p = \mathsf{case}\ p_0\ \mathsf{of}\ \mathit{clauses}'\ \mathsf{end}\ \mathsf{in}\ [\![e]\!]_b^{vs,ps}$$

with $\mathsf{clauses}' = \overline{\mathsf{out}(\text{“}case(vs, ps, l, p_0, p_l, alts)'', [\![e_l]\!]^{vs,ps}_{\mathsf{T}(p)})}$. By the induction hypothesis, we have that $\langle vs, ps\rangle, \theta \cup \sigma \vdash e_i \Downarrow_{\tau_1} p'_i$ implies $[\![e_i]\!]^{vs,ps}_{\mathsf{F}} \Downarrow p'_i$ outputs the sequence of events τ_1. Therefore, $[\![e_i]\!]^{vs,ps}_{\mathsf{T}(p)} \Downarrow p'_i$ outputs and additional event exitcase, and the claim follows by induction.

– Proving the claim for the two remaining rules is straightforward by the induction hypothesis.

A prototype implementation of the program instrumentation can be found at http://kaz.dsic.upv.es/instrument.html. Here, one can introduce a (restricted) Erlang program that is first transformed to the flat syntax and, then, instrumented (several input examples are provided). Moreover, it is also possible to run the instrumented program and obtain the corresponding sequence of events.

5 Concolic Execution

The relevance of the computed sequences of events is that one can easily reconstruct a symbolic execution that mimics the steps of the concrete execution that produced the sequence of events, as well as to produce alternative bindings for the initial variables so that a different execution path will be followed.

Let us first formalize the reconstruction of the symbolic execution from a sequence of events using the Prolog program shown in Fig. 9. As mentioned before, we should ensure that the elements of τ are renamed apart. In our implementation, the sequence of events is written to a file, that is then consulted as a sequence of *facts* and, thus, their variables are always renamed apart. For simplicity, we do not show these low level details in Fig. 9 but just assume the events in τ have been renamed apart.

$sym(\tau, Res, Vars) \leftarrow eval(\tau, [(Res, Vars, BVars)]).$

$eval([\,], [\,]).$

$eval([call(Vars, BVars, NRes, NVars)|Tau], [(Res, Vars, BVars)|Env]) \leftarrow$
$\quad eval(Tau, [(NRes, NVars, NBVars), (Res, Vars)|Env]).$

$eval([case(Vars, BVars, N, Arg, Pat, Alts)|Tau], [(Res, Vars, BVars)|Env]) \leftarrow$
$\quad Arg = Pat,\ eval(Tau, [(Res, Vars, BVars)|Env]).$

$eval([exitcase(Vars, BVars, Arg, Pat)|Tau], [(Res, Vars, BVars)|Env]) \leftarrow$
$\quad Arg = Pat,\ eval(Tau, [(Res, Vars, BVars)|Env]).$

$eval([bind(Vars, BVars, Pat1, Pat2)|R], [(Res, Vars, BVars)|Env]) \leftarrow$
$\quad Pat1 = Pat2, eval(R, [(Res, Vars, BVars)|Env]).$

$eval([exit(Vars, BVars, Pat)|Tau], [(Res, Vars, BVars)|Env]) \leftarrow$
$\quad Res = Pat,\ eval(Tau, Env).$

Fig. 9. Prolog procedure for symbolic execution

Let us briefly explain the rules of the procedure. The first clause just calls *eval* and initializes an stack of function environments with (Res, $Vars$, $BVars$), where

Res is the result of the evaluation, *Vars* are the variables of the main function, and *BVars* are the bounded variables of the main function. When calling *sym*, all these three variables are unbound.

The first rule of $eval/2$ just finishes the computation when there are no events to be processed.

The next rule deals with *call* events and just pushes a new environment (*NRes*,*NVars*,*NBVars*) into the stack of environments. Observe that the names of variables *Vars* and *BVars* occurs twice in the head of the clause—as arguments of the event and as in the current environment—which makes them unify and thus set the right values for them in the current symbolic execution. This is done in all the clauses.

The next rule deals with *case* events and it main purpose is to unify *Arg* and Pat, which represent the case argument and the selected pattern, respectively.

The next rule takes an *exitcase* event and proceeds similarly to the previous one by matching Arg and Pat, now denoting the pattern of a let expression and the result of the evaluation of a case branch.

The next rule deals with a *bind* event in the obvious way by unifying the given patterns Pat_1 and Pat_2.

Finally, the last rule matches *Res* in the current environment (used to store the output of the current function call) with the pattern *Pat* and, moreover, pops the environment (*Res*, *Vars*, *BVars*) from the stack of environments.

For example, given the sequence of events of Example 2 and the initial call $sym(\tau,Res,Vars)$, the above program returns:

$$Res = [X, X], \;\; Vars = [X]$$

which obviously produces less instantiated values than the concrete execution (where we had Res = [a,a], $Vars$= $[a]$).

For concolic testing, though, one is not interested in computing the symbolic execution associated to the concrete execution, but in alternative symbolic executions so that the produced data will give rise to different concrete executions. Luckily, it is easy to extend the previous procedure in order to compute alternative symbolic executions by just replacing the clause for *case* events as follows:

$$\begin{array}{l} eval([case(Vars, BVars, N, Arg, Pat, Alts)|Tau], [(Res, Vars, BVars)|Env]) \\ \quad \leftarrow member((M, Arg', Pat'), Alts), \\ \qquad N \neq M, \;\; Arg' = Pat', \\ \qquad eval(Tau, [(Res, Vars, BVars)|Env]). \end{array}$$

By using the call $member((M, Arg', Pat'), Alts)$, this rule nondeterministically chooses all the alternative selections in case expressions, thus producing alternative bindings for the initial call. For instance, for the sequence of events of Example 2, we get three (nondeterministic) answers:

$$Vars = [\,] \;; \quad Vars = [X] \;; \quad Vars = [X, Y|R]$$

An implementation of the concolic testing tool has been undertaken. The first stage, flattening and instrumenting the source program has been implemented in Erlang itself, and can be tested at http://kaz.dsic.upv.es/instrument.html. In contrast, the concolic testing algorithm is being implemented in Prolog, since the facilities of this language—unification and nondeterminism—make it very appropriate for dealing with symbolic executions.

6 Discussion

In this paper, we have introduced a transformational approach to concolic execution that is based on flattening and instrumenting the source program—a simple first order, eager functional language—. The execution of the instrumented program gives rise to a stream of events that can then be easily processed in order to compute the variable bindings of the associated symbolic executions, as well as possible alternatives. To the best of our knowledge, our paper proposes the first approach to concolic execution by program instrumentation in the context of functional (or logic) programming. In contrast to using an interpreter-based design, in our approach the instrumented program can be run in any environment, even non-standard ones, which opens the door, for instance, to run the instrumented program in a model checking environment like Concuerror [6] so that its execution would produce the sequences of events for all relevant interleavings.

As a future work, we plan to extend our approach in order to cover a larger subset of Erlang as well as to design a fully automatic procedure for concolic testing (currently, one should manually run the instrumented program and the Prolog procedure for generating alternative bindings). Here, we expect that our transformational approach will be useful to cope with concurrent programs, as mentioned above.

Acknowledgements. We thank the anonymous reviewers and the participants of LOPSTR 2015 for their useful comments to improve this paper.

References

1. Armstrong, J., Virding, R., Williams, M.: Concurrent programming in ERLANG. Prentice Hall, Englewood Cliffs (1993)
2. Carlsson, R.: An Introduction to Core Erlang. In: Proceedings of the PLI 2001 Erlang Workshop (2001). http://www.erlang.se/workshop/carlsson.ps
3. Giantsios, A., Papaspyrou, N.S., Sagonas, K.F.: Concolic testing for functional languages. In: Proceedings of PPDP 2015, pp. 137–148. ACM (2015)
4. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: Proceedings of PLDI 2005, pp. 213–223. ACM (2005)
5. Godefroid, P., Levin, M.Y., Molnar, D.A.: Sage: whitebox fuzzing for security testing. Commun. ACM **55**(3), 40–44 (2012)

6. Gotovos, A., Christakis, M., Sagonas, K.F.: Test-driven development of concurrent programs using Concuerror. In: Rikitake, K., Stenman, E. (eds.), Proceedings of the 10th ACM SIGPLAN Workshop on Erlang, pp. 51–61. ACM (2011)
7. Kahn, G.: Natural semantics. In: Brandenburg, F.-J., Vidal-Naquet, G., Wirsing, M. (eds.), Proceedings of STACS 1987, pp. 22–39 (1987)
8. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976)
9. Mesnard, F., Payet, E., Vidal, G.: Concolic testing in logic programming. Theor. Pract. Logic Program. **15**, 711–725 (2015)
10. Pasareanu, C.S., Rungta, N.: Symbolic PathFinder: symbolic execution of Java bytecode. In: Pecheur, C., Andrews, J., Di Nitto, E. (eds.), ASE, pp. 179–180. ACM (2010)
11. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: Proceedings of ESEC/SIGSOFT FSE 2005, pp. 263–272. ACM (2005)
12. Vidal, G.: Towards symbolic execution in Erlang. In: Voronkov, A., Virbitskaite, I. (eds.) PSI 2014. LNCS, vol. 8974, pp. 351–360. Springer, Heidelberg (2015)
13. Vidal, G.: Concolic execution and test case generation in Prolog. In: Proietti, M., Seki, H. (eds.) LOPSTR 2014. LNCS, vol. 8981, pp. 167–181. Springer, Heidelberg (2015)