Dynamic Symbolic Execution of Distributed Concurrent Objects

Andreas Griesmayer, Bernhard Aichernig, Einar Broch Johnsen and Rudolf Schlatte

March 18, 2009

UNU-IIST Report No. 408

$\boxed{\mathcal{R}}$

The United Nations University

UNU/IIST

International Institute for
Software Technology

P.O. Box 3058
Macau

Dynamic Symbolic Execution of Distributed Concurrent Objects

Andreas Griesmayer, Bernhard Aichernig, Einar Broch Johnsen and Rudolf Schlatte

Abstract

This report extends dynamic symbolic execution to distributed and concurrent systems. Dynamic symbolic execution can be used in software testing to systematically identify equivalence classes of input values and has been shown to scale well to large systems. Although usually restricted to sequential programs, this scalability makes it interesting to consider the technique in the distributed and concurrent setting as well. In order to extend the technique to concurrent systems, it is necessary to obtain sufficient control over the scheduling of concurrent activities to avoid race conditions. Creol, a modeling language for distributed concurrent objects, solves this problem by abstracting from a particular scheduling policy but explicitly defining scheduling points. This provides sufficient control to apply the technique of dynamic symbolic execution for model based testing of interleaved processes. The technique has been formalized in rewriting logic, executes in Maude, and applied to non-trivial examples, including an industrial case study.

Andreas Griesmayer is postdoctoral research fellow working on the project CREOL funded by the European Union (IST-33826).
`Email: agriesma@iist.unu.edu`

Bernhard Aichernig is associate research fellow at UNU-IIST and assistant professor at Graz University of Technology.
`Email: bka@iist.unu.edu`

Einar Broch Johnsen is associate professor at University of Oslo.
`Email: einarj@ifi.uio.no`

Rudolf Schlatte is research fellow at UNU-IIST and PhD student at Graz University of Technology, working on the project CREOL funded by the European Union (IST-33826).
`Email: rschlatte@iist.unu.edu`

Contents

1 Introduction

Distributed and concurrent systems, e.g. web services, are becoming increasingly important for long-running infrastructure and applications. They typically consist of loosely coupled components which communicate asynchronously, potentially running on different hardware systems. For critical distributed systems, the use of formal methods, both for design and verification, remains a challenge. In the general case, the complexity of such systems makes full verification seem impossible, even for medium sized examples. In this paper we consider model-based testing of distributed concurrent systems.

We present a tool which identifies adequate test cases from a formal model. In order to test the different communication patterns, we focus on architectural models which reflect the distributed nature of the systems under test. Hence, the models themselves are complex in the sense that they have to capture distribution, concurrency, and asynchronous communication. The challenge is to find a test generation technique that scales to the combinatorial explosion in the number of possible runs in such models. A promising technique that seems to scale well to large systems is *dynamic symbolic execution* [2, 7, 14, 15]. The idea is to calculate a symbolic execution in parallel with the concrete test run of a given formal model. The result is a set of conditions over symbolic input values representing the path of the last run. The conjunction of these conditions form the equivalence class of inputs that could take the same path.

The problem is that dynamic symbolic execution cannot deal with common concurrency models as present in today's programming languages. The reason is that dynamic symbolic execution does not work in the context of arbitrary non-deterministic interleavings of executions. Hence, its main application so far has been limited to single-threaded (sequential) programs and to client-server applications with simple serialized communication flows. In this work we overcome this limitation by choosing a modeling language that provides the appropriate level of concurrency control: Creol [9].

Creol is an executable object oriented modeling language whose execution model was designed to assist in the development of distributed systems. An object in Creol describes an execution unit that executes a dynamic number of processes, a single process at a time. Features like asynchronous method calls and conditional release points allow to model complex interactions between distributed components or objects.

We have implemented the dynamic symbolic execution technique in Maude [3], which is the execution platform of Creol, allowing us to perform the symbolic run dynamically while the concrete run is executed. The tool computes the equivalence classes of test inputs covering the paths already taken, allowing the tester to systematically find new test stimuli for non-covered parts. The generated test cases are used to check the conformance of implementations of the concurrent systems with their Creol models as presented in previous work in [1]. The presented technique forms part of a new design process for distributed systems that has been developed in the EU FP6 CREDO project. It has been applied to the ASK system, an industrial distributed agent-based information system.

To summarize, the *contributions* of this work are as follows:

- This is the first time dynamic symbolic execution is applied to non-trivial concurrent systems involving non-deterministic scheduling of interleaved processes.
- The technique has been formalized in terms of rewriting logic and implemented in the Maude rewriting system.
- It has been applied to an industrial case study.

In the remainder of this section we give an overview of related work, followed by a short introduction to dynamic symbolic execution in the next section and an introduction to Creol in Section 3. Dynamic symbolic execution is extended to concurrent systems in Section 4 and applied to testing in Section 5, before showing examples in Section 6. Finally, in Section 7 we draw our conclusions.

1.1 Related Work

Symbolic execution is a widely used program analysis technique that represents the values of variables as symbolic expressions instead of concrete data. An execution of a program is performed by manipulating those expressions instead of computing concrete values. Application of symbolic execution to testing was already proposed in 1976 by King [11], who shows symbolic execution for a simple sequential language and presents an interactive tool EFFIGY to traverse the execution tree.

Much more recently, symbolic execution has been used for various applications in the area of testing. Khurshid et al. [10] perform source to source transformation on Java programs to allow explicit state model checkers like the Java PathFinder [16] to exploit the succinct representation of the state space by symbolic representation. They generate test cases by checking the reachability of a testing criterion. Analysis of the counter example gives the input for test cases similar to [17, 6, 8]. In [18], Xie et al. introduce SYMSTRA, a tool that uses symbolic execution to explore different sequences of method calls in order to generate unit tests for object oriented systems. These applications use symbolic execution mainly to compress the representation of the state space while performing an exhaustive search. However, there are limits to the feasibility of executing complex concurrent systems purely symbolically, due to the sheer number of possible execution paths induced by non-determinism.

There are basically two possibilities to make the process feasible for large systems: (1) reducing the amount of information which needs to be tracked and (2) reducing the number of paths to search. An example for the first kind are static analysis tools like ARCHER from Engler et al. [19], which very successfully concentrate on certain properties of interest for the analysis (memory and array access). To derive input values that drive a run to certain areas in the program, however, we want to consider all information available. We therefore reduce the number of paths that are searched at the same time to make symbolic execution feasible. The latter technique is called *dynamic symbolic execution.*

To our knowledge, the first to use symbolic execution on single runs were Boyer et al. in 1975 [2] who developed the interactive tool SELECT that computes input values for a run selected by the user. One of the first automated tools for testing was DART (Directed Automated Random Testing) from Godefroid et al. [7]. DART automatically extracts a program's interface and generates a test driver to perform random testing. While DART only evaluates integer variables, the CUTE and jCUTE tools from Sen at al. [14] extend this approach to include pointers and generate dynamic data structures. Several extensions to these approaches exist, among the most notable the PEX tool from Tillmann et al. [15] for creating *parameterized unit tests* for single-threaded .NET programs.

All these approaches use symbolic execution to derive runs on the implementation to detect assertion violations or program crashes, but show very limited support for concurrent programs. In contrast, we develop a model-based approach which targets distributed and concurrent systems and deals with interacting processes and asynchronous communication between components. We extend dynamic symbolic execution to Creol's concurrency model, including the treatment of local scheduling points in the concurrent objects.

There are a number of techniques that help in testing of concurrent systems by either controlling the scheduling to make the test results more deterministic [12] or by repeating test cases multiple times with a different (randomized) scheduling to gain a good coverage of the code [5]. These methods are complementary to the approach shown here as they handle the actual test execution rather then the computation of test cases. Our cases are tailored to check the conformance between an implementation and a model and can be combined with those methods for test execution. Our previous work [1] shows how to use a Creol model as an oracle for a test run on the implementation.

2 An Introduction to Dynamic Symbolic Execution

This section gives a brief introduction to dynamic symbolic execution (DSE) and its application to conventional test case generation, before we proceed with extensions for distributed and concurrent systems. Conventional symbolic execution uses symbols to represent arbitrary values during execution. When encountering a conditional branch statement, the run is forked. This results in a tree covering all paths in the program. In contrast, dynamic symbolic execution calculates the symbolic execution *in parallel* with a concrete run that is actually taken, avoiding the usual problem of eliminating infeasible paths. Decisions on branch statements are recorded, resulting in a set of conditions over the symbolic values that have to evaluate to *true* for the path to be taken. We call the conjunction of these conditions the *path condition*; it represents an equivalence class of concrete input values that could have taken the same path. Note, in the case of non-determinism, there is no guarantee that all inputs will take this path. For the application of DSE to systematic test case generation, the symbolic values represent the inputs of a program; concrete input values from *outside* this equivalence class are selected to force new execution paths, and thereby new test cases.

Consider the following piece of code from an agent system calculating the number of threads needed to handle job requests (taken from Figure 3).

```
amountToCreate:= tasks − idlethreads + ... ;
if (amountToCreate > (maxthreads − threads)) then
  amountToCreate:= maxthreads − threads;
end;
if (amountToCreate > 0) then ... end;
```

Testers usually analyze the control flow in order to achieve a certain coverage. For example, a run evaluating both conditions above to **true** is sufficient to ensure *statement coverage*. *Branch coverage* needs two cases at least and *path coverage* all four combinations. The symbolic computation calculates all possible conditions, expressed in terms of symbolic input values. We denote the symbolic value of an input parameter by appending $_\mathcal{S}$ to the parameter's variable name. Let `threads`, `idlethreads`, and `tasks` denote the input parameters for testing, and `maxthreads` being a constant. Then statement coverage (both conditions evaluate to **true**) is obtained for all input values fulfilling the condition

$$(\mathit{tasks}_\mathcal{S}\text{-}\mathit{idlethreads}_\mathcal{S})>(\mathtt{maxthreads}\text{-}\mathit{threads}_\mathcal{S})$$
$$\wedge(\mathit{maxthreads}_\mathcal{S}\text{-}\mathit{threads}_\mathcal{S})>0$$

Dynamic symbolic execution calculates these input conditions for a concrete execution path. The next test case is generated in such a way that the same path is avoided by negating the input conditions of the previous paths and choosing new input values satisfying this new condition. For example, inputs satisfying

$$(\mathit{tasks}_\mathcal{S}\text{-}\mathit{idlethreads}_\mathcal{S})\leq(\mathtt{maxthreads}\text{-}\mathit{threads}_\mathcal{S})$$
$$\wedge(\mathtt{maxthreads}\text{-}\mathit{threads}_\mathcal{S})>0$$

will avoid the first **then**-branch, resulting in a different execution path.

One immediately realizes that the choice of which sub-condition to negate determines the kind of coverage obtained. Note that the coverage that actually can be achieved depends on the actual program and the symbolic values used. For example, the presence of unreachable code obviously makes full statement coverage impossible. The concrete test values from symbolic input vectors can be found by, e.g., using a constraint solver.

3 The Modeling Language Creol

Creol is a high-level executable modeling language targeting distributed systems in which concurrent objects communicate asynchronously [9]. The language decouples communication from synchronization. Furthermore, it allows local scheduling to be left underspecified but controlled through explicitly declared process release points. The language has a formal semantics defined in rewriting logic [13] and executes on the Maude platform [3]. In the remainder of this section, we present Creol and point out its essential features for DSE.

A concurrent object in Creol executes a number of processes that have access to its local state. Each process corresponds to the activation of one of the object's methods; a special method `run` is automatically activated at object creation time, if present, and captures the object's active behavior. Objects execute concurrently: each object has a processor dedicated to executing the processes of that object, so processes in different objects execute in parallel. In contrast to, e.g., Java, each Creol object strictly encapsulates its state; i.e., external manipulation of the object state happens via calls to the object's methods only.

Only one process can be active in an object at a time; the other processes in the object are *suspended*. We distinguish between *blocking* a process and *releasing* a process. Blocking causes the execution of the process to stop, but does not let a suspended process resume. Releasing a process suspends the execution of that process and lets another (suspended) process resume. Thus, if a process is blocked there is no execution in the object, whereas if a process is released another process in the object may execute. The execution of several processes within an object can be combined using *release points* within method bodies. At a release point, the active process may be released and *some* suspended process resumes. This way, (non-terminating) active and reactive behavior are easily combined within a concurrent object in Creol.

Communication in Creol is based on method calls. These are a priori asynchronous; method replies are assigned to labels (also called *future variables*, see [4]). There is no synchronization associated with *calling* a method. *Reading a reply* from a label, however, is a blocking operation and allows the calling object to synchronize with the callee. A method call that is directly followed by a read operation models a synchronous call. Thus, the calling process may decide at runtime whether to call a method synchronously or asynchronously. The local scheduling of processes inside an object is given by conditions associated with release points. These conditions may depend on the value of the local state, allowing cooperative scheduling between the processes within an object, but may also depend on the object's communication with other objects in the environment. Guards on release points include synchronization operations on labels, so the local scheduling can depend on both the object's state and the arrival of replies to asynchronous method calls.

To sum up: only one process is executing on each object's local state at a time, and interleaving of processes is flexibly controlled via (guarded) release points. Together with the fact that objects communicate exclusively via messages (strict encapsulation), this gives us the concurrency control necessary for extending DSE to the distributed paradigm.

Syntax. The language syntax of the subset of Creol used in this paper is presented in a Java-like style in Figure 1. In this overview, we omit some features of Creol, including interfaces, inheritance, non-deterministic choice and many built-in data types and their operations. For a full overview of Creol, see for example [9]. In the language subset used in the examples

$$
\begin{array}{rll@{\qquad}rll}
T & ::= & C \mid \texttt{\textbf{Bool}} \mid \texttt{\textbf{Void}} & L & ::= & \texttt{\textbf{class}}\ C(\overline{v})\ \textbf{begin}\ \overline{\textbf{var}\ f:T};\overline{M}\ \textbf{end} \\
 & \mid & \texttt{\textbf{Int}} \mid \texttt{\textbf{String}} \mid \ldots & M & ::= & \textbf{op}\ m(\textbf{in}\ \overline{x:T}\ \textbf{out}\ \overline{x:T}) == \overline{\textbf{var}\ x:T};\overline{s}\ \textbf{end} \\
v & ::= & f \mid x & e & ::= & v \mid \textbf{new}\ C(\overline{v}) \mid \texttt{\textbf{null}} \mid \texttt{\textbf{this}} \mid v+v \mid \ldots \\
b & ::= & \texttt{\textbf{true}} \mid \texttt{\textbf{false}} \mid v & s & ::= & l!e.m(\overline{e}) \mid !e.m(\overline{e}) \mid l?(v) \mid e.m(\overline{e};\overline{v}) \mid \texttt{\textbf{await}}\ g \\
g & ::= & b \mid v? \mid g \wedge g & & \mid & v := e \mid \texttt{\textbf{skip}} \mid \texttt{\textbf{release}} \mid \texttt{\textbf{await}}\ e.m(\overline{e};\overline{v}) \\
 & & & & \mid & \texttt{\textbf{while}}\ g\ \textbf{do}\ \overline{s}\ \textbf{end} \mid \texttt{\textbf{if}}\ g\ \textbf{then}\ \overline{s}\ \textbf{end}
\end{array}
$$

Figure 1: Language syntax of a subset of Creol.

of this paper, classes L are of type C with a set of methods $\overline{M}$.*Expressions* e over variables v (either fields f or local variables x) are standard. *Statements* s are standard apart from the asynchronous method call $l!e.m(\overline{e})$ where the label l points to a reference to the reply, the (blocking) read operation $l?(\overline{v})$, and release points **`await`** g and **`release`**. *Guards* g are conjunctions of Boolean expressions b and synchronization operations $l?$ on labels l. When the guard in an **`await`** statement evaluates to *false*, the statement is *disabled* and becomes a **`release`**, otherwise it is *enabled* and becomes a **`skip`**. A **`release`** statement suspends the active process and another suspended process may be rescheduled. The *guarded call* **`await`** $e.m(\overline{e};\overline{v})$ is a typical pattern which suspends the active process until the reply to the call has arrived and abbreviates $l!e.m(\overline{e});$**`await`** $l?;l?(\overline{v})$.

3.1 Representation of a Run

A run of a Creol system captures the parallel execution of processes in different concurrent objects. Such a run may be perceived as a sequence of execution steps where each step contains a set of local transitions on a subset of the system's objects. However, only one process may be active at a time in each object and different objects operate on disjoint data. Therefore, the transitions in each execution step may be performed in a truly concurrent manner or in any sequential order, so long as all transitions in one step are completed before the next execution step commences. For the purposes of dynamic symbolic execution the run is represented as a sequence of statements which manipulate the state variables, together with the conditions which determine the control flow, as follows.

The representation of an assignment $\overline{v} := \overline{e}$ is straightforward: Because fields and local variables in different processes can have the same name and statements from different objects are interleaved, the variable names are expanded to a unique identifier by adding the object id for fields and the call label for local variables. This expansion is done transparently for all variables and we will omit the variable scope in the following.

An asynchronous method call in the run is reflected in four execution steps (the label value l uniquely identifies the steps that belong to the same method call): $o_1 \overset{l}{\rightharpoonup} o_2.m(\overline{e})$ represents the *call* of method m in object o_2 from object o_1 with arguments $\overline{e}$; $o_1 \overset{l}{\rightharpoondown} o_2.m(\overline{v})$ represents when the called objects starts execution, where $\overline{v}$ are the local names of the parameters for m; $o_1 \overset{l}{\leftharpoonup} o_2.m(\overline{e})$ represents the emission of the return values from the method execution; and $o_1 \overset{l}{\leftharpoondown} o_2.m(\overline{v})$ represents the corresponding reception of the values. These four events fully describe method calling in Creol. In this execution model the events reflecting a specific method call always appear in the same order, but they can be interleaved with other statements.

Object creation, **new** $C(\overline{v})$, is similar to a method call. The actual object creation is reduced to generating a new identifier for the object and a call to the object's **init** and **run** methods, which create the sequences as described above.

Conditional statements in Creol are side effect free and therefore only represented in form of the statements of the branch that was actually executed. For the sake of computing the input values, however, the condition of the taken branch is recorded as $\langle g \rangle$. Remark that statements **await** g requires careful treatment: if it evaluates to *false*, no code is executed. To reflect the information that the interpreter failed to execute a process because the condition g of the **await** statement evaluated to *false*, the negated condition $\langle \neg g \rangle$ is recorded.

4 Dynamic Symbolic Execution of Concurrent Objects

This section presents the rules to actually compute the symbolic values for a given run. The formulas given in this section very closely resemble the rewrite rules of Creol's simulation environment [9], defined in rewriting logic [13] and implemented in Maude [3]. A rewrite rule $t \Longrightarrow t'$ may be interpreted as a *local transition rule* allowing an instance of the pattern t in the configuration of the rewrite system to evolve into the corresponding instance of the pattern t'. When auxiliary functions are needed in the semantics, these are defined in equational logic, and are evaluated in between the state transitions [13]. The rules are presented here in a slightly simplified manner to improve readability.

Denote by $\overline{s}$ the representation of program statements. Let $\sigma = \langle v_1 \rhd e_1, v_2 \rhd e_2, \dots v_n \rhd e_n \rangle = \langle \overline{v} \rhd \overline{e} \rangle$ be a map which records *key–value* entries $v \rhd e$, where a variable v is bound to a symbolic value e. The value assigned to key v is accessed by $v\sigma$. For an expression e and a map σ, define a parallel substitution operator $e\sigma$ which replaces all occurrences of every variable v in e with the expression $v\sigma$ (if v is in the domain of σ). For simplicity, let $\overline{e}\sigma$ denote the application of the parallel substitution to every expression in the list $\overline{e}$. Furthermore, let the operator $\sigma_1 \uplus \sigma_2$ combine two maps σ_1 and σ_2 such that, when entries with the same key exist in both maps, the entry in σ_2 is taken. These operators are defined as equations in rewriting logic and are evaluated in between the rewrite steps. In the symbolic state σ, all expanded variable names are bound to symbolic expressions. However, operations for method calls do not change the value of the symbolic state, but generate or receive *messages* that are used to communicate actual parameter values between the calling and receiving objects. Similar to the expressions bound to variables in the symbolic state σ, the symbolic representations of these actual parameters are bound in a map Θ to the actual and unique label value l provided for each method call by Creol's operational semantics. Finally, the conditions of control statements along an execution path are collected in a list $\mathcal{C}$; the concatenation of a condition c to $\mathcal{C}$ is denoted by $\mathcal{C}\hat{}c$.

The *configurations* of the rewrite system for dynamic symbolic execution are given by $\overline{s}\big[\Theta, \sigma, \mathcal{C}\big]$,

$$\begin{aligned}
\overline{v} := \overline{e}; \overline{s}\big[\Theta, \sigma, \mathcal{C}\big] &\Longrightarrow \overline{s}\big[\Theta, \sigma \uplus \langle \overline{v} \rhd (\overline{e}\sigma)\rangle, \mathcal{C}\big] && (\textsc{assign})\\
o_1 \overset{l}{\rightharpoonup} o_2.m(\overline{e}); \overline{s}\big[\Theta, \sigma, \mathcal{C}\big] &\Longrightarrow \overline{s}\big[\Theta \uplus \langle l \rhd \overline{e}\sigma\rangle, \sigma, \mathcal{C}\big] && (\textsc{call})\\
o_1 \overset{l}{\rightharpoondown} o_2.m(\overline{v}); \overline{s}\big[\Theta, \sigma, \mathcal{C}\big] &\Longrightarrow \overline{s}\big[\Theta, \sigma \uplus \langle \overline{v} \rhd l\Theta\rangle, \mathcal{C}\big] && (\textsc{bind})\\
\langle g\rangle; \overline{s}\big[\Theta, \sigma, \mathcal{C}\big] &\Longrightarrow \overline{s}\big[\Theta, \sigma, \mathcal{C}\,\hat{}\,\langle g\sigma\rangle\big] && (\textsc{cond})
\end{aligned}$$

Figure 2: Rewrite rules for symbolic execution of Creol statements.

where $\overline{s}$ is a run represented as a sequence of statements, Θ and σ are the maps for messages and symbolic variable assignments as described above, and $\mathcal{C}$ is the list of conditions. Recall that the run $\overline{s}$ (as described in Section 3.1) is in fact generated on the fly by the concrete rewrite system for Creol executed in parallel with the dynamic symbolic execution. Thus, the *rules* of the rewrite system have the form

$$\overline{s}\big[\Theta, \sigma, \mathcal{C}\big] \Longrightarrow \overline{s}'\big[\Theta', \sigma', \mathcal{C}'\big]$$

The primed versions are updated results from the execution rule. The rules are given in Figure 2 and explained below.

Rule ASSIGN defines the variable updates that are performed for an assignment. All variables in the right hand side are replaced by their current values in σ, which is then updated by the new expressions. Note that we do not handle variable declarations, but work in the runtime-environment. We expect that a type check already happened during compile time and insert variables into σ the first time they appear. A method `call` as defined by Rule CALL emits a message that records the expressions that are passed to the method. Because of the asynchronous behavior of Creol, the call might be received at a later point in the run (or not at all if the execution terminates before the method was selected for execution) by Rule BIND, which handles the binding of a call to a new process and assigns the symbolic representation of the actual parameter values to the local variables in the new process. The emission and reception of return values are handled similarly to call statements and call reception.

Object creation is represented as a call to the constructor method `init` of the newly created object. In this case there is no explicit label for the call statement, so the object identifier is used to identify the messages to call the `init` and `run` methods, which are associated to the `new` statement. For conditionals, the local variables in the condition are replaced by their symbolic values (Rule COND). This process is identical for the different kinds of conditional statements (**`if`**, **`while`**, **`await`**). The statement itself acts as a **`skip`** statement; it changes no variables and does not produce or consume messages. The resulting expression $g\sigma$ directly characterizes the equivalence class of input values that reach and fulfill the condition. The conjunction of all conditions found during symbolic evaluation give the set of input values that can perform that run. The tool records the condition that evaluated to *true* during runtime. Therefore, if the **`else`** branch of an **`if`** statement is entered or a disabled **`await`** statement with g approached, the recorded condition will be $\neg g$.

5 Application to Testing

Approaches to test case generation for structural coverage intend to find test sets that perform runs in the system for a specific coverage criterion. Two runs that cover the same parts of a system can be considered equivalent. A good test set should maximize the coverage, while minimizing the number of equivalent runs in order to avoid superfluous efforts in executing the tests.

The execution of a concurrent system is not fully controllable through its interface. One and the same test case can lead to arbitrarily different runs on the system under test (SUT). In practice, tools like ConTest [5] are used to execute single test cases multiple times on the SUT with different schedulings. For the model, on the other hand, it is straightforward to introduce additional variables to resolve the nondeterminism for the sake of examining all possible paths to build the optimal set of test cases. These techniques are complementary to the computation shown in this paper and should be applied additionally.

It is the responsibility of a testing engineer to write test objects (analogous to unit tests) that set up the system and perform interactions that will drive an interesting execution of the system. Presupposing this test scenario, we enhance the coverage by introducing symbolic values $\mathtt{t}_{\mathcal{S}}$ in the test object and compute new values such that new, non-equivalent runs are performed.

Constructing the Test Set. Dynamic symbolic execution on a run gives the set of conditions that are combined to the path condition $\mathcal{C} = \bigwedge_{1 \leq i \leq n} c_i$ (for n conditions), characterizing exactly the equivalence class of $\mathtt{t}_{\mathcal{S}}$ that can repeat the same execution path. Only one test case that fulfills $\mathcal{C}$ is required. A new test case is then chosen to specifically avoid that a particular branch is taken by violating the respective c_i. To maximize decision coverage (DC), for instance, test cases have to be created such that for each of the conditions c_i, there is also a test case that violates this condition. The process of generating new test cases ends after all combinations required for the coverage criteria are explored.

In the case of concurrent distributed systems, however, we frequently deal with scenarios in which the naive approach does not terminate. Most importantly, concurrent systems usually contain active objects that do not terminate and thus creates an infinite run. In this case, execution on the model has to be stopped after exceeding some threshold (ideally after detecting a loop). The computation of the condition can be performed as before and will prohibit the same partial run in future computations. Creol also supports infinite datatypes. Therefore, for a code sample like `while (i > 0) do i := i - 1 end`, there is a finite run for each `i`, but there are infinitely many of them. To make sure that the approach terminates, an artificial limiting condition has to be introduced, e.g., by creating an equivalence class for all `i` greater than a constant `k`.

Running a Test Case. A test case as generated in this paper is used to test implementations of concurrent systems by checking if the implementation under test complies to the model as described in previous work [1]. The test execution approach of that paper handles the difficulties of testing a concurrent system by defining a set of actions and events that are used to control the implementation as well as the model, and to monitor the behavior of the implementation. So far the execution has not been monitored online, rather a log is generated that has been verified by using the model. A run of the implementation is considered successful if the model is able to reproduce the run.

The model is a direct specification of the implementation, and both systems share their internal control structure. Test cases optimized for structural coverage in the model will therefore also improve the structural coverage in the implementation.

6 Examples

This section shows the feasibility of the approach by means of two examples: The *peer to peer* example presents the exploration of existing test cases with respect to coverage, during which an important special case was discovered. The second example demonstrates how to derive new test cases on example of the *ASK system*, an industrial case study.

The dynamic symbolic interpreter allows to identify variables that are treated as normal variables for the concrete run, and as a symbolic value for the dynamic symbolic execution. These variables are identified by a special naming scheme, here denoted by the subscript $_{\mathcal{S}}$. This enables the flexible monitoring of symbolic values of variables at any arbitrary level in the code. This is in contrast to PEX [15] which uses parameterized unit tests.

6.1 Peer to Peer

A peer to peer system connects several coequal components (peers) with the aim to share data between them. Each peer works both as client and as server holding local files. A client can search the network to find the location of a file, connect to the respective server and download the document. Communication between the components is established via channels. We use a sophisticated model describing such a system, which is stems from the CREDO project to demonstrate various techniques for modeling of distributed systems. It consists of 23 classes (not shown in this paper due to lack of space) and already comes with a small set of test cases that model a net consisting of three nodes with some files each. User interaction is modeled by a class `Tester` that communicates with one of the Peers and simulates communication from the user interface. In the following example, Class `Tester` models an interaction sequence where the user searches for a file document named `"f1"` and the result is stored in the variable `reply`.

```
class Tester(cl :Client, b :Peer) contracts User
begin
  var reply  :Data
  op run == await cl.search("f1";reply)
end
```

Symbolic evaluation is used to check the paths this test case creates by replacing the constant `"f1"` by $reqkey_{\mathcal{S}}$, with the effect that the assignment in Line 6 is only executed to generate the concrete run, the symbolic execution passes the symbolic value $reqkey_{\mathcal{S}}$ to the method `cl.search`.

```
1 class Tester_new(cl :Client, b :Peer) contracts User
2 begin
3   var reply  :Data
4   op run ==
5     var reqkey_S  :Data;
6     reqkey_S:= "f1" ;
7     await cl.search(reqkey_S;reply)
8 end
```

Running our tool on the example gives us two decisions that depend on $reqkey_{\mathcal{S}}$:

```
{"ifthenelse" : not( in(reqkey_S, ["f2"])) }
{"ifthenelse" : in( reqkey_S, ["f1"]) }
```

The called server contacts its peers and checks the stored list of files if `"f1"` is present. The first server does not contain the key `"f1"` but file `"f2"` only. (Condition 1), but a check at the second server is successful (Condition 2). Examination of all predefined test cases showed that this pattern repeats for each test case. For proper coverage we are interested in concrete values of $reqkey_{\mathcal{S}}$ not satisfying the already taken decisions. In our example this means that we need a value executing a path that does not end in finding a file. Hence, a new concrete value (e.g. `"f0"`) that is not contained in any of the three servers was assigned to $reqkey_{\mathcal{S}}$, what led to the following path condition:

```
{"ifthenelse" : not(in(reqkey_S,["f2"])) }
{"ifthenelse" : not(in(reqkey_S,["f1"])) }
{"ifthenelse" : not(in(reqkey_S,["f1", "f2", "f3"])) }
```

This new test case represents the important case that ensures that all servers are contacted and the client performs properly even if no file was found.

6.2 The ASK System

ASK is an industrial software system for connecting and organizing people, developed by the research company Almende and marketed by ASK Community Systems. The ASK system provides mechanisms for matching users requiring information or services with potential suppliers and is used by various organizations for applications like workforce planning and emergency response. The number of people connected varies from several hundred to several thousands.

A Creol reference model for ASK systems has been developed by Almende [1]. The ASK system consists of a number of components to receive and process requests. Each of these components is itself multi-threaded. The threads inside a component act as workers in a thread pool, the

```
1  op createThreads ==
2    var amountToCreate : Int;
3    var idlethreads : Int:= threads − busythreads;
4    await ((threads< maxthreads)
5         ∧ ((idlethreads − tasks)< (threads/ 2)));
6    amountToCreate:= tasks − idlethreads +(threads/ 2);
7    if (amountToCreate > (maxthreads − threads)) then
8      amountToCreate:= maxthreads − threads;
9    end;
10   if (amountToCreate > 0) then
11     await threadpool.createThreads(amountToCreate);
12   end;
13   createThreads();
```

Figure 3: Model of thread pool balancing code in the ASK system. The fields `threads`, `idlethreads` and `tasks` are updated by outside method calls, so the conditions in the **`await`** statements can become true.

executing tasks are put into a component-wide shared task queue. A *balancer* is used to create and destroy worker threads depending on a given maximal number of threads, the currently existing number of threads and on the number of remaining tasks. Figure 3 shows one central part of this balancing task: the tail-recursive method `createThreads`. This method and its opponent in the model, `killThreads`, are responsible for creating and killing threads when appropriate. The balancer is initialized with the symbolic value $\mathtt{maxthreads}_S$, the maximum number of threads that are allowed in the thread pool. Inside the balancer, the local variable `maxthreads` is then set to $\mathtt{maxthreads}_S$ `+ 1` to account for the balancer thread itself, which also runs inside the thread pool. The balancer has access to the number of threads that are active (`threads`), the number of threads that are processing some task (`busythreads`), and the number of tasks that are waiting to be assigned to a worker thread (`tasks`).

The **`await`** statement in Line 4 suspends the process if it is not necessary to create further worker threads, i.e. if the maximal number of threads is already reached or half of the threads are without a task (they are neither processing a task, nor is there a task open for processing). The **`if`** statement in Line 7 makes sure there are not more tasks created than allowed by `maxthreads`. Finally, the thread pool is ordered to create the required numbers of threads in Line 11.

We instantiate the model with a fixed number of tasks (10 in our example) and with a variable maximum of threads $\mathtt{maxthreads}_S$, with the goal of finding different values for $\mathtt{maxthreads}_S$ to optimize the coverage of the code in Figure 3. In the following, we show only the relevant parts of the calculated path conditions, leaving out conditions pertaining to other parts of the model (`killThreads`, the thread creation code inside `threadpool`, etc.).

For a first run we choose $\mathtt{maxthreads}_S$`==0`. Dynamic symbolic execution with this starting value results in the path condition:

```
{"disabled␣await" : not( 1< (maxthreads_S +1) & true) }
```

After a little simplification it becomes clear that the path was taken because `0 >=` $\mathtt{maxthreads}_S$.

Any other start value will lead to a different run. A start value $\mathtt{maxthreads}_S$==15 generates

```
{"enabled␣await" : (1< (maxthreads_S +1) & true) }
{"ifthenelse" : not(10 > maxthreads_S ) }
```

The number 10 reflects the number of tasks we created. The path condition reflects that all inputs with $\mathtt{maxthreads}_S$ `>= 10` lead to the same path because in each case only the number of threads is created, which is 10 due to the 10 tasks with which the model was initialized. There is no condition for the **`if`** in Line 10 because the amount to create does not exceed $\mathtt{maxthreads}_S$ and therefore is not dependent on it. A third run, created with $\mathtt{maxthreads}_S$==5, results in

```
{"disabled␣await" : (1< (maxthreads_S +1) & true) }
{"ifthenelse" : 10 > maxthreads_S }
{"ifthenelse" : maxthreads_S > 0 }
```

In this test case the amount of tasks to create exceeded the maximal allowed number of tasks and therefore was recomputed in Line 8. The new value depends on $\mathtt{maxthreads}_S$, which causes the **`if`** statement in Line 10 to contribute to the path condition. The new path condition does not further divide the input space, so the maximal possible coverage according to the chosen coverage criterion is reached.

7 Conclusions

The main contribution of this work is the novel extension of dynamic symbolic execution to non-trivial distributed and concurrent object models. This has been achieved by exploiting the properties of the Creol modeling language; in particular local scheduling control of the processes and strict encapsulation of the object state. This paper demonstrates how dynamic symbolic execution, combined with the executable architectural models of Creol, can be used to systematically derive interesting test cases, while avoiding the combinatorial explosion inherent in distributed concurrent systems. Our approach has been formalized in rewriting logic and implemented in Maude. A peer to peer example and an industrial case study of an agent system serve to illustrate the technique.

The current version of the tool reports the equivalence classes to the user, but does not automatically select and execute new test runs. Immediate future work will be an automation of this process by means of constraint solving techniques. Others have shown that this is feasible in practice, e.g. in [15].

Dynamic symbolic execution, as presented in this paper, should be applicable to other object-oriented languages with concurrency by enforcing serialization of processes in the object as well as strict encapsulation. In a multi-threaded concurrency model as found in Java, dynamic symbolic execution could in principle be achieved by declaring all methods as synchronized and all fields as private. However, such severe restrictions seem undesirable. It would be interesting if lighter restrictions for such languages could be identified that still enable dynamic symbolic execution.

References

[1] B. Aichernig, A. Griesmayer, R. Schlatte, and A. Stam. Modeling and testing multi-threaded asynchronous systems with Creol. In *Proceedings of the 2nd International Workshop on Harnessing Theories for Tool Support in Software (TTSS'08)*, ENTCS. Elsevier, 2009. To appear.

[2] R. S. Boyer, B. Elspas, and K. N. Levitt. Select-A formal system for testing and debugging programs by symbolic execution. *SIGPLAN Not.*, 10(6):234–245, 1975.

[3] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, Aug. 2002.

[4] F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In R. de Nicola, editor, *Proc. 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *LNCS*, pages 316–330. Springer, Mar. 2007.

[5] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for testing multi-threaded Java programs. *Concurrency and Computation: Practice & Experience*, 15(3):485–499, 2003.

[6] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In *ESEC/FSE-7: Proc. of the 7th European software engineering conference*, volume 1687 of *LNCS*, pages 146–162. Springer-Verlag, 1999.

[7] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223. ACM, 2005.

[8] H. Hong, I. Lee, O. Sokolsky, and H. Ural. A temporal logic based theory of test coverage and generation. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *LNCS*, pages 327–341. Springer, 2002.

[9] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.

[10] S. Khurshid, C. Pasareanu, and W. Visser. Generalized Symbolic Execution for Model Checking and Testing. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, volume 2619 of *LNCS*, pages 553–568. Springer, 2003.

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

[12] B. Long, D. Hoffman, and P. A. Strooper. Tool Support for Testing Concurrent Java Components. *IEEE Trans. on Software Engineering*, pages 555–566, 2003.

[13] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[14] K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *Proc. 18th International Conference on Computer Aided Verification (CAV'06)*, volume 4144 of *LNCS*, page 419. Springer, 2006.

[15] N. Tillmann and J. de Halleux. Pex - white box test generation for .NET. In *Proc. of the 2nd International Conference on Tests and Proofs (TAP'08)*, volume 4966 of *LNCS*, pages 134–153. Springer, 2008.

[16] W. Visser, K. Havelund, G. Brat, and S. Park. Java PathFinder - second generation of a Java model checker. In *Proc. of Post-CAV Workshop on Advances in Verification, Chicago, July*, 2000.

[17] W. Visser, C. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 97–107. ACM New York, NY, USA, 2004.

[18] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, volume 3440 of *LNCS*, pages 365–381. Springer, 2005.

[19] Y. Xie, A. Chou, and D. Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 327–336. ACM, 2003.