

Conformance Testing of Distributed Concurrent Systems with Executable Designs ^{*}

Bernhard K. Aichernig^{1,2}, Andreas Griesmayer¹,
Einar Broch Johnsen³, Rudolf Schlatte^{1,2}, and Andries Stam⁴

¹ International Institute for Software Technology, United Nations University
(UNU-IIST), Macao S.A.R., China
{agriesma,bka,rschlatte}@iist.unu.edu

² Institute for Software Technology, Graz University of Technology, Austria
{aichernig,rschlatte}@ist.tugraz.at

³ Department of Informatics, University of Oslo, Norway
einarj@ifi.uio.no

⁴ Almende BV, The Netherlands
andries@almende.org

Abstract. This paper presents a unified approach to test case generation and conformance test execution in a distributed setting. A model in the object-oriented, concurrent modeling language Creol is used both for generating test inputs and as a test oracle. For test case generation, we extend Dynamic Symbolic Execution (also called Concolic Execution) to work with multi-threaded models and use this to generate test inputs that maximize model coverage. For test case execution, we establish a conformance relation based on trace inclusion by recording traces of events in the system under test and replaying them in the model. User input is handled by generating a test driver that supplies the needed stimuli to the model. An industrial case study of the Credo project serves to demonstrate the approach.

Keywords. Model-based testing, conformance testing, concolic execution, Creol, Maude.

1 Introduction

Model-based testing has become an increasingly important part of robust software development practices. Specifying a system's behavior in a formal model helps to uncover specification ambiguities that would otherwise be resolved in an ad-hoc fashion during implementation. Using the model as a test oracle as well as a specification aid reinforces its critical role in the development process.

The techniques presented in this paper are based on the object-oriented modeling language Creol, a language designed to model concurrent and distributed

^{*} This research was carried out as part of the EU FP6 project *Credo*: Modeling and analysis of evolutionary structures for distributed services (IST-33826).

systems. Creol models are high-level as they abstract from, e.g., particular network properties as well as specific local schedulers. However, Creol is an executable language with a formal semantics defined in rewriting logic [19]. Thus, Creol models may be seen as executable designs. Test cases are written in Creol as well, and *dynamic symbolic execution* (DSE) is applied to calculate a test suite that reaches the desired model coverage. DSE is a combination of concrete and symbolic execution, and therefore, it is also known as *concolic execution*.

To show conformance between model and implementation, sequences of events are recorded from the instrumented implementation and replayed on the model. This approach allows reasoning about control flow and code coverage and goes beyond observations on program input/output. The conformance relation is based on trace inclusion, that is, every behavior shown by the implementation must be observable on the model as well. In case of non-deterministic models, we apply model-checking techniques in order to reach conclusive fail verdicts. To deal with user input events, the generated test driver stimulates the model in the same way as was observed in the implementation.

This testing methodology is applied in the context of the ASK system, one of the industrial demonstrators of the Credo project. However, Creol and the presented model-based testing technique is general and covers a wide range of distributed architectures.

The major contributions of this paper are:

- A tool-supported method for calculating optimal-coverage test cases from a model that serves as a test oracle.
- An extension of DSE to deal with concurrency.
- A conformance relation that can handle both input/output events and internal actions in a uniform way and allows reasoning about program flow and code coverage.
- A tool to generate a test driver from recorded implementation behavior that copes with arbitrary input events.

The rest of the paper is organized as follows: Section 2 gives an in-depth overview of the approach, including the techniques and the conformance relation developed, and Section 3 presents the unique features of the Creol modeling language that enable parallel DSE. Section 4 explains how to calculate test inputs from a Creol model, and Section 5 shows how to generate full test cases by recording an implementation’s behavior responding to these test inputs, and checking whether its Creol model can exhibit the same behavior. Finally, Sections 6 and 7 contain related work and a conclusion to the paper.

2 Overview: the Testing Approach

The method described in this paper consists of two parts: generating test cases from a Creol model, and validating the implementation against the model. *Generating test cases* is done by computing test input values to achieve maximal

model coverage. To handle the parallelism in the models, dynamic symbolic execution is used to avoid the combinatorial state space explosion that is inherent in static analysis of such systems. *Validating the implementation* is achieved via light-weight instrumentation of both model and implementation, and replaying traces that were recorded on the implementation on the model in order to verify the conformance of the implementation’s behavior.

2.1 Finding Test Cases with Dynamic Symbolic Execution

This section gives a brief introduction to dynamic symbolic execution (DSE) and its application to test case generation of sequential programs. Our extensions for distributed and concurrent systems are presented in Section 4.1. 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 of this equivalence class 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. Hence, the selection of new input values for finding new paths is a typical constraint solving problem.

Example 1. Consider the following piece of code from an agent system calculating the number of threads needed to handle job requests.

```

1  amountToCreate:= tasks - idlethreads + ... ;
2  if (amountToCreate > (maxthreads - threads)) then
3     amountToCreate:= maxthreads - threads;
4  end;
5  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 $(tasks_{\mathcal{S}} - idlethreads_{\mathcal{S}}) > (maxthreads - threads_{\mathcal{S}}) \wedge (maxthreads_{\mathcal{S}} - threads_{\mathcal{S}}) > 0$

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

$$(tasks_S - idlethreads_S) \leq (\maxthreads - threads_S) \\ \wedge (\maxthreads - threads_S) > 0$$

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, but the coverage that can actually be achieved also 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 modern constraint solvers (e.g., ILOG Solver) or SMT-solvers (e.g., Yices, Z3).

2.2 Conformance Testing Using Recorded Event Traces

In the setting of asynchronous, concurrent systems, and when facing non-determinism, testing for expected behavior by examining the outputs of the *system under test* (SuT) is not always sufficient. Our approach utilizes the observed structural similarity of a model written in Creol and its implementation to test that the implementation has a similar control flow as the executable model. To this end, both model and implementation are *instrumented* at points in the code where meaningful *events* occur. At a high level, an implementation can be seen as a mapping I from an initial configuration $conf_I$ to an event trace $events_I$ – or more generally, in the face of nondeterminism, to a set of event traces $\{events_I\}$. Similarly, the instrumented model M maps an initial configuration $conf_M$ to a set of traces $\{events_M\}$.

Given a function ρ that converts (refines) configurations from the model to the implementation view, and a function α to abstract event traces from implementation to the model, the relationship between model and implementation can be seen in Diagram 1:

$$\begin{array}{ccc} conf_M & \xrightarrow{M} & \{events_M\} \\ \downarrow \rho & & \uparrow \alpha \\ conf_I & \xrightarrow{I} & \{events_I\} \end{array} \quad (1)$$

In the literature this is also called U-simulation [12]. The conformance relation of the approach can then be described as follows: given a test input (written by a test engineer or calculated via DSE), all possible event traces resulting from stimulating the implementation by that input must also be observable on the model. Equation 2 shows the formulation of this trace inclusion relation:

$$\alpha(I(\rho(conf_M))) \subseteq M(conf_M) \quad (2)$$

Section 5.2 shows an implementation of the α function as a generated Creol test driver class that is run in parallel with the instrumented model to reach a test verdict. Some of the recorded events correspond to user input to the implementation; the generated test driver supplies the equivalent stimuli to the model.

In contrast to the automated α mapping, currently the ρ mapping between initial configurations is manual.

3 Creol

Creol is a modeling language for executable designs, targeting distributed systems in which concurrent objects communicate asynchronously [18]. 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 [19] and executes on the Maude platform [9]. In the remainder of this section, we present Creol and emphasize 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 represents 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. Note, due to the non-deterministic scheduling semantics of Creol it is possible that the active process may be immediately rescheduled for execution. Hence, (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 [10]). There is no synchronization associated with *calling* a method. However, *reading a reply* from a label 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.

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

Fig. 1. Language syntax of a subset of Creol.

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.

In summary, only one process is executing on each object’s local state at a time, and the 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 Figure 1. In this overview, we omit some features of Creol, including inheritance, non-deterministic choice, and many built-in data types and their operations. For a full overview of Creol, see for example [18]. In the language subset used in the examples of this paper, classes L are of type C with a set of methods \bar{M} . Classes can implement zero or more interfaces, which define methods that the class must then implement. *Expressions* e over variables v (either fields f or local variables x) are standard. *Statements* s are standard apart from the asynchronous method call $!e.m(\bar{e})$ where the label l points to a reference to the reply, the (blocking) read operation $l?(\bar{v})$, 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. Hence, the suspended process releases lock on the object’s attributes. The *guarded call* **await** $e.m(\bar{e}; \bar{v})$ is a typical pattern which suspends the active process until the reply to the call has arrived and abbreviates $!e.m(\bar{e}); \mathbf{await} \ l?; l?(\bar{v})$.

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 atomic execu-

tion steps where each step contains a set of local state-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 $\bar{v} := \bar{e}$ 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 unique identifiers 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 sequel.

An asynchronous method call in the run is reflected in four execution steps (remark that the label value l uniquely identifies the steps that belong to the same method call): $o_1 \xrightarrow{l} o_2.m(\bar{e})$ represents the *call* of method m in object o_2 from object o_1 with arguments \bar{e} ; $o_1 \xrightarrow{l} o_2.m(\bar{v})$ represents the moment when a called object starts execution, where \bar{v} are the local names of the parameters for m ; $o_1 \xleftarrow{l} o_2.m(\bar{e})$ represents the emission of the return values from the method execution; and $o_1 \xleftarrow{l} o_2.m(\bar{v})$ 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** $C(\bar{v})$, 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, i.e. they do not change an object’s state. In order to record the choice made during a run, the condition or its negated version are included into the run as Boolean guard $\langle g \rangle$. Hence, a run represents both, the variable changes together with the taken branch. As will be shown later, the conditions in a run are used to calculate the equivalence class of all input values that may take this path.

Await statements **await** g require careful treatment: if they evaluate 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.

3.2 The ASK Case Study

We demonstrate the approach using the ASK system as a running example throughout the paper. ASK is an industrial software system for connecting people based on context-aware response, developed by the research company Al-mende [3] and marketed by ASK Community Systems [4]. The ASK system

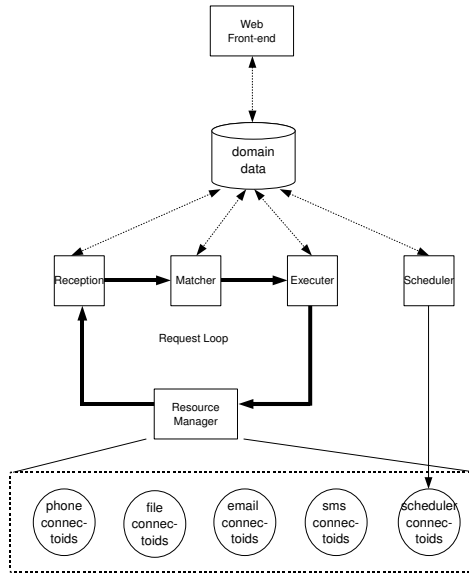


Fig. 2. Overview of the ASK system architecture.

provides mechanisms for matching users requiring information or services with potential suppliers and is used by various organizations for applications such as workforce planning and emergency response. The number of people connected varies from several hundred to several thousand, making it imperative to have good testing support.

Figure 2 shows the basic architectural view of the existing ASK systems. The “heartbeat” of the system is the *Request loop*, indicated with thick arrows. A request contains the information of two *participants* (a requester and a responder). Based on the request, the ASK system attempts to provide a connection between the two participants if possible, and otherwise attempts to suggest an alternative responder. A number of more or less independent components work together to search for appropriate participants for a request, determine how to connect them, or in general figure out the best way in which a request can be fulfilled.

Each of these components is itself multi-threaded. The threads act as workers in a thread pool, executing tasks put into a component-wide shared task queue. Tasks are used to implement the requests described above. Within a single component, threads do not communicate directly with each other. However, they can dispatch new tasks to the task queue that are eventually executed by another or the same thread. Threads are also able to send messages to other components. In most of the components, the number of threads can change over time, depending on the number of pending tasks in the task queue and on the number of idle threads.

```

1 class ThreadPool(size : Int, maxNofThreads : Int)
2   contracts ThreadPool
3 begin
4   vars taskCtr, threadCtr, busyCtr :Counter;
5   var taskQueue : TaskQueue;
6   var threads : List[Thread];
7   var balancer : Task;
8
9   op init ==
10  // removed variable initialization
11  balancer:=new BalancerTask(1, taskCtr, threadCtr, busyCtr,
12    maxNofThreads, mrate, taskQueue, this);
13  this.dispatchTask(balancer;)
14
15  with Any op dispatchTask(in task : Task) ==
16  taskQueue.enqueueTask(task;)
17
18  with Any op createThreads(amount : Int) ==
19  var i : Int;
20  var thread : Thread;
21  i:= 0;
22  while (i< amount) do
23    thread:=new Worker(taskQueue, busyCtr, threadCtr);
24    threads:=threads  $\vdash$  thread; // append thread
25    threadCtr.inc(;);
26    i:= i +1
27  end
28
29  with Any op start ==
30  this.createThreads(size;)
31 end

```

Fig. 3. ThreadPool of the ASK system (instantiation of Counter and TaskQueue omitted).

A reference model for ASK systems has been developed in Creol, in collaboration with Almende [2]. An example of a class in Creol is given in Figure 3, which shows the implementation of the ThreadPool and also contains the system-wide task queue. The thread pool is initialized with the parameters `size` and `maxNofThreads` which determine the initial number of threads in the pool and the maximum allowed number respectively. It also contains a number of counters to keep record of tasks and threads (number of pending tasks `taskCtr`, total number of worker threads `threadCtr`, and number of worker threads that currently execute a task `busyCtr`). The initialization of these variables is straightforward and omitted in the shown code for matters of presentation. When the class is initialized, the `init` method is automatically executed and creates the *balancer* task which is responsible for creating and deleting working

threads when needed. We will discuss this thread in more detail in Section 4.2. The dispatch method inserts tasks into the task queue to be executed by an idle worker thread. Method `createThreads` creates a given number of worker threads, which themselves look into the task queue for open tasks. Note that in Creol input and output parameters are separated by semicolon. Hence, the absence of output is indicated by a semicolon at the end of the actual parameter list, as e.g. in the call to `createThreads` at the end. When the system is set up, the thread pool is activated by the `start` method (being called from a client object), which calls `createThreads` with the initial number of worker threads (as set by the class parameter `size`).

This reference model forms the basis for our work on testing ASK systems. Note that in this paper we only show excerpts of the model and omit some of the details for better demonstration of the approach. This simplified model consist of six different kinds of objects with various instances and does not induce any performance problems.

4 Test Case Generation

To generate test cases from the Creol model, we extend dynamic symbolic execution from Section 2.1 to distributed concurrent objects. Coverage criteria define a measurement of the amount of the program that is covered by the test suite. Two runs that cover the same parts of a system can be considered equivalent. A good test suite maximizes the coverage while minimizing the number of equivalent runs in order to avoid superfluous effort in executing the tests.

To set up a test case, the testing engineer first selects a test scenario, a description of the intention of the test, either from use cases or a high level specification of the system. Using this scenario, a first test run is set up that triggers a corresponding execution of the system. Starting with this run, the coverage is enhanced by introducing symbolic values t_S in the test object and computing new values such that new, non-equivalent runs are performed.

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 t_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 by violating some c_i so that another branch is executed. Note that by executing new branches, also new conditions may be discovered. To reach decision coverage (DC) in a test suite, for instance, test cases are created until for each condition c_i there is at least one test case that reach and fulfill as well as violate this condition. The process of generating new test cases ends after all combinations required for the required coverage criterion are explored.

In the case of distributed concurrent systems, however, we frequently deal with scenarios in which the naive approach does not terminate. Most importantly, such concurrent systems often contain active objects that do not terminate and thus create an infinite run. In this case, execution on the model has to be stopped after exceeding some threshold. The computation of the path condi-

tion can be performed as before and will prohibit the same partial run in future computations. Creol also supports infinite datatypes. For a code sample such as **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, a limiting condition has to be introduced manually, for example by creating an equivalence class for all i greater than a user defined constant.

4.1 Dynamic Symbolic Execution in the Parallel Setting

We now present the rules to compute the symbolic values for a given run. The formulas given in this section very closely resemble the rewrite rules of the Creol simulation environment [18], defined in rewriting logic [19] and implemented in Maude [9]. 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 [19]. The rules are presented here in a slightly simplified manner to improve readability.

Denote by \bar{s} the representation of a sequence of program statements. Let $\sigma = \langle v_1 \triangleright e_1, v_2 \triangleright e_2, \dots, v_n \triangleright e_n \rangle = \langle \bar{v} \triangleright \bar{e} \rangle$ be a map which records *key-value* entries $v \triangleright e$, where a variable v is bound to a symbolic value e . The value assigned to the 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 $\bar{e}\sigma$ denote the application of the parallel substitution to every expression in the list \bar{e} . Furthermore, let the expression $\sigma_1 \uplus \sigma_2$ combine two maps σ_1 and σ_2 so that, when entries with the same key exist in both maps, the entry in σ_2 is taken. 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}^c .

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

$$\bar{s}[\Theta, \sigma, \mathcal{C}] \Longrightarrow \bar{s}'[\Theta', \sigma', \mathcal{C}'].$$

$$\begin{aligned}
\bar{v} := \bar{e}; \bar{s}[\Theta, \sigma, \mathcal{C}] &\Longrightarrow \bar{s}[\Theta, \sigma \uplus \langle \bar{v} \triangleright (\bar{e}\sigma) \rangle, \mathcal{C}]. && \text{(ASSIGN)} \\
o_1 \xrightarrow{l} o_2.m(\bar{e}); \bar{s}[\Theta, \sigma, \mathcal{C}] &\Longrightarrow \bar{s}[\Theta \uplus \langle l \triangleright \bar{e}\sigma \rangle, \sigma, \mathcal{C}]. && \text{(CALL)} \\
o_1 \xrightarrow{l} o_2.m(\bar{v}); \bar{s}[\Theta, \sigma, \mathcal{C}] &\Longrightarrow \bar{s}[\Theta, \sigma \uplus \langle \bar{v} \triangleright l\Theta \rangle, \mathcal{C}]. && \text{(BIND)} \\
\langle g \rangle; \bar{s}[\Theta, \sigma, \mathcal{C}] &\Longrightarrow \bar{s}[\Theta, \sigma, \mathcal{C} \langle g\sigma \rangle]. && \text{(COND)}
\end{aligned}$$

Fig. 4. Rewrite rules for symbolic execution of Creol statements.

The primed terms on the right-hand side are updated results from the execution of the rule. The rules are given in Figure 4 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 expression $g\sigma$ characterizes the equivalence class of input values that fulfill the condition if it is reached. The conjunction of all conditions found during symbolic evaluation represents the set of input values that can trigger 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 is approached, the recorded condition will be $\neg g$.

4.2 The ASK Case Study Revisited

We revisit our running example to demonstrate the parallel version of DSE and the way test cases are generated. The *balancer* Task is instantiated by the ThreadPool in Figure 3 to compute the number of worker threads to create or

```

1  op init ==
2    maxthreads:=maxthreads+1;
3
4  op createThreads ==
5    var amountToCreate : Int;
6    var idlethreads : Int:= threads - busythreads;
7    await ((threads<maxthreads)
8            $\wedge$  ((idlethreads - tasks) < (threads/2)));
9    amountToCreate:= tasks - idlethreads + (threads/2);
10   if (amountToCreate > (maxthreads - threads)) then
11     amountToCreate:= maxthreads - threads;
12   end;
13   if (amountToCreate > 0) then
14     await threadpool.createThreads(amountToCreate);
15   end;
16   createThreads();// infinite loop by tail-recursion

```

Fig. 5. Parts of the balancing thread to initialize and create new threads. The fields `threads`, `idlethreads` and `tasks` are updated by outside method calls, so the conditions in the `await` statements can become true.

destroy depending on a given maximal number of threads, the currently existing number of threads and the number of remaining tasks. Figure 5 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 as needed. The balancer is initialized with $maxthreads_S$, the maximum number of threads that are allowed in the thread pool. In the balancer's `init` method (not shown here), the local variable `maxthreads` is incremented by one to account for the balancer task 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 the line 7 suspends the process while 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 10 checks that there are not more tasks created than allowed by `maxthreads`. Finally, the thread pool is instructed to create the required numbers of threads in the line 14.

Figure 6 shows the code to instantiate the model and create a fixed number of tasks (10 in our example). The dynamic symbolic interpreter allows to treat special *variables* as *values*. Such variables are treated as a symbolic value for the dynamic symbolic execution and are selected by a special naming scheme, here denoted by the subscript S . This enables a flexible monitoring of symbolic values of variables at any arbitrary level in the code.

```

1 class Main(nthreadsS : Int, maxWorkThreadsS : Int)
2 begin
3   var threadpool : ThreadPool;
4   var executionCounter : Counter;
5
6   op init ==
7     threadpool := new ThreadPool(nthreadsS, maxWorkThreadsS);
8     executionCounter := new Counter;
9
10  op run ==
11    var task : Task;
12    var i : Int;
13    i := 0;
14    while (i < 10) do
15      task := new CounterTask(i, executionCounter);
16      threadpool.dispatchTask(task);
17      i := i + 1;
18    end
19    threadpool.start();
20    // After running, the executionCounter should be 10
21 end

```

Fig. 6. Setting up a model for DSE. Here, *nthreads_S* is the number of initial threads to be created and *maxthreadss* is the maximal size of the thread pool.

The test case setup of Figure 6 uses two symbolic variables as parameters: the maximum number of working threads *maxWorkThreads_S* and the initial number of threads *nthreads_S*. DSE is used to find different concrete values for those symbolic values to optimize the coverage of the model.

For a first run we randomly choose the initial values *maxWorkThreads_S*==0 and *nthreads_S*=1. Dynamic symbolic execution with these starting values results in the path condition:

```

{"ifthenelse" : (0 < nthreadsS) }
{"ifthenelse" : not (1 < nthreadsS) }
{"disabled_await" : not ( 1 < (maxWorkThreadsS + 1) ∧ true) }

```

The first two conditions are from the loop in the line 22 of Figure 3 and correspond to one loop traversal in which a thread is created. The third condition corresponds to the line 7 in Figure 5 and shows that the path was taken because $0 \geq \text{maxWorkThreads}_S$ and the balancer is not allowed to create any worker threads. Any other start values will lead to a different run.

Each of the conjuncts in the path condition depends only on the input *maxWorkThreads_S*. For easier presentation, we will exploit this fact in the following and compute new values only for this input and leave *nthreads_S* constant. Note that this is generally not the case, conditions that rely on sev-

eral symbolic values require that the input space is partitioned considering all variables.

For the second run we choose a value that is outside the previously computed path condition and continue with $maxWorkThreads_S == 15$, which records the conditions:

```
{ "enabled_await" : (1 < (maxWorkThreads_S + 1) ^ true) }
{ "ifthenelse" : not (10 > maxWorkThreads_S ) }
```

for the **await** in line 7 and the **if** in line 10 of Figure 5. The number 10 in the second condition reflects that we create ten tasks at initialization in Figure 6. The path condition reflects that all inputs with $maxWorkThreads_S \geq 10$ lead to the same path because there will not be more threads created than the number of outstanding tasks. There is no condition for the **if** in the line 13 because the amount to create does not exceed $maxWorkThreads_S$ and therefore is not dependent on it.

A third run, created with $maxWorkThreads_S == 5$, results in

```
{ "disabled_await" : (1 < (maxWorkThreads_S + 1) ^ true) }
{ "ifthenelse" : 10 > maxWorkThreads_S }
{ "ifthenelse" : maxWorkThreads_S > 0 }
```

In this test case the amount of threads to create exceeded the maximal allowed number of threads and therefore was recomputed in line 11. The new value depends on $maxWorkThreads_S$, which causes the **if** statement in the line 13 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.

The $nthreads_S$ variable controls the initial number of threads in the threadpool, and is the only variable that determines the number of traversals through the loop in the line 22 of Figure 3. This is also reflected in the path condition that we got from $nthreads_S == 1$ — it states that the same path through the loop will be taken if $(0 < nthreads_S)$ and $(1 \geq nthreads_S)$, i.e., $nthreads_S == 1$. Thus, using this condition for test case selection, we need a test case for each value of $nthreads_S$, it is not possible to create bigger equivalence classes. A closer look at the path condition shows us how to create a new run that never traverses the loop: negating the first condition, $(0 < nthreads_S)$. Thus, we get a new test case with $nthreads_S == 0$ (we keep the value $maxWorkThreads_S == 5$ from the initial test case). The path condition only consists of:

```
{ "ifthenelse" not (0 < nthreads_S) }
```

None of the conditions of Figure 5 is reached. This is due to the fact that in this case no worker thread is created on initialization of the threadpool, thus, the balancer cannot be executed.

Test cases with $nthreads_S > 1$ lead to similar test cases as the initial one, with the variation that a different number of threads are calculated to be created. If too many threads are created in the beginning, the tasks are all completed before the balancer is called. This is because the tasks in the model are strongly

```

<trace>
  <createThreads thread="3079972528" time="501911878"
    number="10"/>
  <starting thread="3075214224" time="501911929" info=""/>
  <waiting thread="3075214224" time="501911951" info=""/>
  <starting thread="3066821520" time="501911980" info=""/>
  <waiting thread="3066821520" time="501911999" info=""/>
  ...
  <enqueue thread="3079972528" time="501912403"
    info="Sabbey - balancer (Sabbey.c 353)"/>
  ...
</trace>

```

Fig. 7. Parts of a recorded event trace from the ASK system. At the beginning, 10 threads are created; each thread emits a `starting` and a `waiting` event when created. Later, a task is added to the system.

abstracted versions of the real implementation and complete instantly. A delay in the tasks or more tasks in the test setup can be used to solve that problem.

The computation of the values for $maxWorkThreads_S$ can be automatized by constraint- or SMT solvers. For the example above we used Yices [11], which takes the negated path condition as input and computes an valuation for the variables if it is satisfiable.

5 Test Case Execution

Section 4 explained how to calculate test inputs for the implementation. This section describes how to reach test verdicts by generating test drivers to run the test cases and validate the implementation’s behavior against the model. As mentioned in Section 2, our test assumption is that a sequence of events that is observed on the implementation can be reproduced (replayed) by the model.

5.1 Obtaining Traces from the Implementation

In order to obtain traces of events, the implementation is instrumented via code injection. The case study, where the system under test is implemented in C, uses AspectC [5] for this purpose; similar code injection or aspect-oriented programming solutions can be used for systems implemented in other languages.

Traces are recorded in a simple XML-based format, for ease of automatic processing. Figure 7 shows parts of a trace from the ASK system. At the start, a `createThreads` event occurs, followed by the events associated with threads being started and waiting for a task to work on (`starting` and `waiting`, respectively). Other events used in the case study are `killThreads` (recorded when the balancing thread decides to remove some threads), `enqueue` (recorded when a new Task is created) and `dequeue` (recorded when a thread starts working on a task).

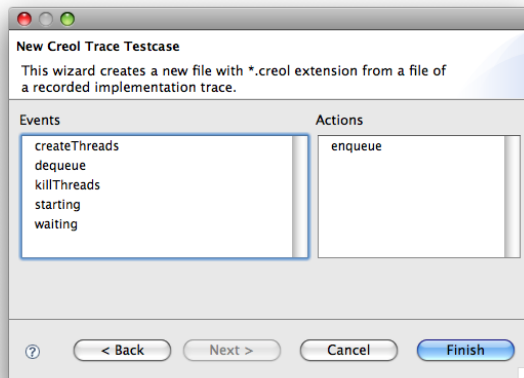


Fig. 8. Generating the tester from a recorded trace: separating Actions and Events. “enqueue” is to be triggered by the tester, so is designated to be an Action using this dialog.

5.2 Generating the Test Driver

As mentioned in Section 2.2, some of the events recorded in the implementation originate from the environment (user input, incoming network data, etc.). We call these “external” events *actions*, and generate a test driver that stimulates the model in the same way. In the example, enqueue is an event that comes from outside – in the implementation, it is typically triggered by an incoming phone call or by a database-stored work queue; the test driver has to trigger the same action when replaying the trace on the model.

Figure 8 shows the dialog that is used to differentiate actions and events from the recorded trace for the purpose of generating the tester. Each action is a stimulus that the tester gives to the model. The tool generates a Creol interface `TestActions` and a class `TestAdapter` which is ready to contain code for initializing the model and for stimulating the model from the test case implementation. Methods with empty bodies are generated for these purposes.

Figure 9 shows the interface `TestActions` and class `TestAdapter` that are generated using the choices made in Figure 8. The one designated action (“enqueue”) results in a method called `enqueue`, which will be called by the generated tester code. In the `TestAdapter` class, the test engineer then supplies implementations for model initialization (Figure 9, line 9) and any actions (line 11).

In addition to implementing the methods in `TestAdapter`, the test engineer has to add events to the model at the place equivalent to where they were added in the implementation to record the trace. At each point where an event occurs, the model communicates with the tester, indicating which event is about to happen. The thread of execution which generates an event is *blocked* until the tester accepts the event; other threads can continue executing. The tester, in turn, waits for each event in sequence and then unblocks the model so that it can

```

1 interface TestActions
2 begin
3   with Any op enqueue(in thread:Int, time:Int, info:String)
4 end
5
6 class TestAdapter implements TestActions
7 begin
8   op init ==
9     skip// TODO: implement test driver setup here
10  with Any op enqueue(in thread:Int, time:Int, info:String) ==
11    skip// TODO: implement enqueue action
12 end

```

Fig. 9. Test actions interface and test adapter class template, created from the implementation trace.

continue. The model thus synchronizes with the sequence of events recorded from the implementation, as implemented by the tester. The following code snippet shows the `createThreads` event added to the `createThreads` method from Figure 5:

```

if (amountToCreate > 0) then
  tester.request("createThreads"); // EVENT
  await threadpool.createThreads(amountToCreate);
end;

```

Figure 10 shows parts of the tester’s run method; the sequence of Creol statements corresponds one-to-one to the sequence of events and actions in the trace of Figure 7. The `ok` variable is set to `false` at the beginning and to `true` at the end of the run method; this allows us to use model checking to find a successful run. Each action in the trace is converted to a call to the corresponding method in the `TestAdapter` class, as implemented by the user. In line 14 is a call to `action`. Each event is converted to a pair of statements, the first statement (`this.allow(...)`) unblocks the model and allows the event to occur, the second statement (`await get(...)`) blocks the tester until the event actually occurs in the model.

5.3 Obtaining Test Verdicts

To actually run the test case, an instance of the generated `TestCase` class is generated. Its `init` method, inherited from `TestAdapter` and implemented by the user, sets up and starts the model, and its `run` method (Figure 10), generated from the recorded implementation trace, steers the model to generate the expected events in sequence.

A test results in a verdict of “pass” if the model can reproduce the trace recorded from the implementation and if all assertions and invariants in the

```

1  op run ==
2    ok := false;
3    this.allow("createThreads");
4    await get(sem, "createThreads") = 0;
5    this.allow("starting");
6    await get(sem, "starting") = 0;
7    this.allow("waiting");
8    await get(sem, "waiting") = 0;
9    this.allow("starting");
10   await get(sem, "starting") = 0;
11   this.allow("waiting");
12   await get(sem, "waiting") = 0;
13   ...
14   this.enqueue(3079972528, 501912403,
15               "Sabbey_ _balancer_(Sabbey.c_353)");
16   ...
17   ok := true

```

Fig. 10. Replaying the trace of Figure 7: tester event and action behavior in the model.

model hold. If an assertion in the model is violated, the model itself has an inconsistency and is in error (assuming the model is supposed to be valid for all inputs); no verdict about the implementation can be reached. If the run method of `TestCase` runs to completion, the test passes. If the tester deadlocks when running in parallel with the model, the implementation potentially violates the test assumption. But this result is inconclusive, it is still a possibility that a different scheduling in the model allows the test to pass; model checking the combination of model and tester can give a definitive answer and let us reach a verdict of “pass” or “fail”.

6 Related Work

To our knowledge, the first to use symbolic execution on single runs were Boyer et al. in 1975 [8] who developed the interactive tool SELECT that computes input values for a run selected by the user. Some of the first automated tools for testing were DART (Directed Automated Random Testing) from Godefroid et al. [15], and the CUTE and jCUTE tools from Sen et al. [20]. Perhaps the most prominent and most widely used tool in that area is PEX by Tillmann et al. [21], which creates parameterized unit tests for single-threaded .NET programs. A closer look at DSE for generating test input in a parallel setting can be found in [17, 16], recent work on examining all relevant interleavings in [1].

The use of formal models for testing has a long history, some of the more influential work are [14] and [22]. Various conformance relations have been proposed. They place varying demands w.r.t. controllability and observability placed on the SuT; for example *ioco* [23] by Tretmans et al. demands that implemen-

tations be input-enabled, while Petrenko and Yevtushenko's *queued-quiescence testing* does away with that assumption. Our proposed conformance relation is even more permissive, in that arbitrary input can become part of the test case and conforming behavior is checked after the fact instead of in parallel with the implementation.

Most tools for automated or semi-automated model-based software testing, including TorX [6] and TGV [13], work by simulating a user of the system, controlling input and checking output. A testing method similar to the one described in this paper, also relying on event traces, was developed by Bertolino et al. [7], whereby at run-time traces are extracted and model-checked to verify conformance to a stereotyped UML2 model. They emphasize black-box testing of components and reconstruct cause-effect relationships between observed events to construct message sequence charts. Consequently, they have to employ more intrusive monitoring than our approach.

7 Conclusions

We have presented an approach to test case generation and conformance testing which is integrated into the development cycle of distributed systems. We specify models in Creol, a concurrent object-oriented modeling language for executable designs of distributed systems. A single model serves to both optimize test cases in terms of coverage, and as test oracle for test runs on the actual implementation. Test input generation and model coverage are controlled via dynamic symbolic execution extended to a parallel setting, which has been implemented on top of the Maude execution platform for Creol. The conformance relation is based on U-simulation. Only a lightweight level of instrumentation of the implementation is needed, which is here achieved by means of aspect-oriented programming. The problem of reaching conclusive verdicts in case of non-determinism is handled by replaying the traces using Maude's search facilities. The techniques have been successfully applied in the context of the ASK systems, one model serving as a reference for several versions of the system.

Acknowledgments The authors wish to thank the anonymous reviewers for their helpful comments and clarifications.

References

1. B. Aichernig, A. Griesmayer, M. Kyas, and R. Schlatte. Exploiting distribution and atomic transactions for partial order reduction. Technical Report No. 418, UNU-IIST, June 2009.
2. 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.
3. Almende website. <http://www.almende.com>.

4. ASK community systems website. <http://www.ask-cs.com>.
5. ACC: The AspeCt-oriented C compiler. <http://www.aspectc.net>.
6. A. Belinfante, J. Feenstra, R. G. de Vries, J. Tretmans, N. Goga, L. M. G. Feijs, S. Mauw, and L. Heerink. Formal test automation: A simple experiment. In G. Csopaki, S. Dibuz, and K. Tarnay, editors, *12th International Workshop on Testing of Communicating Systems*, volume 147 of *IFIP Conference Proceedings*, pages 179–196. Kluwer, 1999.
7. A. Bertolino, H. Muccini, and A. Polini. Architectural verification of black-box component-based systems. In N. Guelfi and D. Buchs, editors, *Rapid Integration of Software Engineering Techniques (RISE 2006)*, volume 4401 of *LNCS*, pages 98–113. Springer, May 2007.
8. 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.
9. 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.
10. 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.
11. L. de Moura and B. Dutertre. A fast linear-arithmetic solver for DPLL(T). In *Proc. 18th International Conference on Computer Aided Verification (CAV'06)*, volume 4144 of *LNCS*, pages 81 – 94. Springer, Aug. 2006.
12. W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge University Press, 1998.
13. J.-C. Fernandez, C. Jard, T. Jérón, and C. Viho. Using on-the-fly verification techniques for the generation of test suites. In R. Alur and T. A. Henzinger, editors, *Computer Aided Verification, 8th International Conference, CAV '96*, volume 1102 of *LNCS*, pages 348–359. Springer, 1996.
14. M.-C. Gaudel. Testing can be formal, too. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *TAPSOFT '95: Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, volume 915, pages 82–96. Springer, 1995.
15. 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.
16. A. Griesmayer, B. Aichernig, E. Johnsen, and R. Schlatte. Dynamic symbolic execution for testing distributed objects. In *Second International Conference on Tests and Proofs (TAP'09)*, volume 5668 of *LNCS*, pages 105–120. Springer, July 2009.
17. A. Griesmayer, B. Aichernig, E. Johnsen, and R. Schlatte. Dynamic symbolic execution of distributed concurrent objects (short paper). In *IFIP International Conference on Formal Techniques for Distributed Systems (FMOODS/FORTE'09)*, volume 5522 of *LNCS*, pages 225–230. Springer, June 2009.
18. E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.
19. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.
20. 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.

21. 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.
22. J. Tretmans. Test generation with inputs, outputs, and quiescence. In T. Margaria and B. Steffen, editors, *Second Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, volume 1055 of *LNCS*, pages 127–146. Springer, 1996.
23. M. van der Bijl, A. Rensink, and J. Tretmans. Compositional testing with IOCO. In A. Petrenko and A. Ulrich, editors, *Formal Approaches to Software Testing, Third International Workshop on Formal Approaches to Testing of Software (FATES 2003)*, volume 2931 of *LNCS*, pages 86–100. Springer, 2003.