Testing Concurrent Objects with Application-Specific Schedulers*

Rudolf Schlatte [1,2], Bernhard Aichernig [1,2], Frank de Boer [3],
Andreas Griesmayer [1], and Einar Broch Johnsen [4]

[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@ist.tugraz.at

[3] CWI, Amsterdam, Netherlands
frb@cwi.nl

[4] Department of Informatics, University of Oslo, Norway
einarj@ifi.uio.no

Abstract. In this paper, we propose a novel approach to testing executable models of concurrent objects under application-specific scheduling regimes. Method activations in concurrent objects are modeled as a composition of symbolic automata; this composition expresses all possible interleavings of actions. Scheduler specifications, also modeled as automata, are used to constrain the system execution. Test purposes are expressed as assertions on selected states of the system, and weakest precondition calculation is used to derive the test cases from these test purposes. Our new testing technique is based on the assumption that we have full control over the (application-specific) scheduler, which is the case in our executable models under test. Hence, the enforced scheduling policy becomes an integral part of a test case. This tackles the problem of testing non-deterministic behavior due to scheduling.

1 Introduction

In this paper we address the problem of testing executable high-level behavioral models of concurrent objects. In contrast to multi-threaded execution models for object-oriented programs such as, e.g., the Java model for the parallel execution of threads, we consider in this paper a model of object-oriented computation which describes a method call in terms of the generation of a corresponding process in the callee. The concurrent execution of objects then naturally arises from asynchronous method calls, which do not suspend while waiting for the return value from the method calls. Objects execute their internal (encapsulated) processes in parallel. In this setting, the scheduling of the internal processes of

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

an object directly affects its behavior (both its functional and non-functional behavior). Therefore, a crucial aspect of the analysis of concurrent objects is the analysis of the intra-object scheduling of processes. In contrast to scheduling on the operating-system level, the object-level scheduling policies will be fine-tuned according to the application requirements. We call this *application-specific scheduling*. In this paper we introduce a novel testing technique for concurrent objects under application-specific scheduling regimes.

We develop a testing technique for concurrent objects in the context of Creol [9, 4], a high-level modeling language which allows for the abstraction from implementation details related to deployment, distribution, and data types. The semantics of this language is formalized in rewriting logic [11] and executes on the Maude platform [3]. As such the Creol modeling language also allows for the simulation, testing, and verification of properties of concurrent object models, based on execution on the Maude platform as described by formal specifications. One of the main contributions of this paper is a formal testing technique for this language which integrates formal specifications of application-specific scheduling regimes at an abstraction level which is *at least as high as that of the modeling language*. The novelty of this approach is that it takes the scheduling policy as an integral part of a test case in order to control its execution.

In order to specify test cases in our formal testing technique, we first develop suitable behavioral abstractions of the mechanisms for synchronizing the processes within an object, as featured by the modeling language. The integration of these behavioral abstractions and the formal specification of a particular scheduling regime provides the formal basis for the generation of test cases. For the formal specification of test purposes we use assertions which express required properties of the object state (or a suitable abstraction thereof). Test cases are then generated by applying a weakest precondition calculus in order to find an abstract behavior which satisfies the assertions [8]. The execution of a test case on the Maude platform requires instrumenting the Maude interpreter of Creol's operational semantics such that it will enforce the embodied scheduling policy on the processes of the particular concurrent object which is considered by the test case. Particular test cases address the behavior of the concurrent object model under a given, formally defined scheduling regime. If such a test case fails to reach its goal (test purpose), this might indicate a problem with the given scheduling policy. Hence, the relevance of this contribution for modeling object-oriented systems in general is that it also allows the specification and analysis of scheduling issues in an early stage of design, as an integral part of the high-level models. However, in the following discussion we focus on the important aspect of controlling test-case execution by enforcing a scheduling regime.

Paper overview. The rest of this paper is organized as follows: Section 2 introduces the Creol language and executable modeling. Section 3 gives a high-level overview and scope for our approach to testing. Section 4 explains the model-ling approach used, including the high-level specification of scheduling policies. Section 5 discusses the details of test case generation and execution. Finally, Section 6 discusses related work and Section 7 concludes the paper.

$$
\begin{array}{ll}
sr ::= s \mid s; \texttt{return}\ e & L ::= \texttt{class}\ C(\overline{v})\ \{\overline{T\ f}; \overline{M}\} \\
v ::= f \mid x & M ::= T\ m\ (\overline{T\ x})\ \{\overline{T\ x}; sr\} \\
b ::= \texttt{true} \mid \texttt{false} \mid v & e\ ::= v \mid \texttt{new}\ C(\overline{v}) \mid e.\texttt{get} \mid e!m(\overline{e}) \mid \texttt{null} \mid \texttt{this} \mid \texttt{caller} \\
T ::= C \mid \texttt{Bool} \mid \texttt{Void} & s\ ::= v := e \mid \texttt{await}\ g \mid \texttt{skip} \mid s; s \\
g ::= b \mid v? \mid g \wedge g & \quad\ \mid \texttt{if}\ g\ \texttt{then}\ s\ \texttt{fi} \mid \texttt{release}
\end{array}
$$

Fig. 1. The language syntax. Variables v are fields (f) or local variables (x), and C is a class name.

2 Creol and Executable Modeling

In the design of component-based or object-oriented systems, it may be desirable to introduce a separation of concerns between business code, dealing with the functionality of the software unit, and synchronization code, dealing with the local scheduling of different computing activities. Creol is a high-level executable modeling language for concurrent objects in which such scheduling may be left underspecified [9]. The language has a formal semantics defined in rewriting logic [11] and executes on the Maude platform [3]. This allows various analysis techniques to be developed and applied to the Creol models, including, e.g., pseudo-random simulation and breadth-first search through the execution space.

In contrast to, e.g., Java, each Creol object encapsulates its state; i.e., all external manipulation of the object state happens through calls to the object's methods. Each process corresponds to the activation of one of the object's methods. In addition, 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 Creol, method calls are asynchronous and assigned to so-called futures [4]. Only one process may 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. Although processes need not terminate, the execution of several processes within an object may be combined using *release points* within method bodies. Release points may include polling operations on futures, to check for the arrival of replies to asynchronous method calls. At a release point, the active process may be released and *some* suspended process may resume.

Syntax. The language syntax of the subset of Creol used in this paper is presented in a Java-like style in Fig. 1. For the purpose of this paper, we emphasize the differences with Java and focus on the specification of a single class. At present, we omit some features of Creol, including inheritance and method calls. *Expressions* e are standard apart from the asynchronous method call $e!m(\overline{e})$, the (blocking) read operation v.**get**, and the pseudo-variable **caller** which refers

```
class batch_queue(Nat x) {
  Nat wc, batch, comein // waiting clients, barrier size
  Seq[Object] display // queue of registered client objects

  Void batch_queue() { batch := x; wc := 0; comein := 0 }

  Void register() {
    wc := wc+1;
    if wc ≥ batch then comein := batch fi;
    await comein > 0;
    comein := comein - 1;
    wc := wc-1;
    display := (display;caller);
  }
}
```

Fig. 2. Motivating example: The `batch_queue` class.

to the caller of the current method activation. *Statements s* are standard apart from release points **`await`** g and **`release`**. *Guards g* are conjunctions of Boolean expressions b and polling operations $v?$ on futures v. When the guard in an **`await`** statement evaluates to *false*, the statement becomes a **`release`**, otherwise a **`skip`**. A **`release`** statement suspends the active process and another suspended process may be rescheduled.

Example. We consider a version of barrier synchronization given by the class `batch_queue` in Fig. 2.. In a `batch_queue` object, clients are processed in batches (of size `batch`, the parameter `x` to the constructor sets the size of the batches). A client which registers must wait until enough clients have registered before getting assigned a slot in the queue. For simplicity, we represent the queue as a local variable `display`, which is a sequence of clients (semicolon is the append operator on sequences). Before any call to `register` will return, the object will contain `batch` processes.When enough calls are waiting to be registered, the next batch of processes may proceed by assigning the value of `batch` to `display`. It is easy to see that the order in which callers are added to the display sequence depends on the internal scheduling of processes in the object.

Once more, we mention that only a subset of Creol is presented in this paper; the interested reader is referred to e.g. [9].

3 Testing and Testing Methodology

The executable formal semantics of the Creol language allows the application of different analysis techniques. In this section we briefly sketch our proposed methodology for testing Creol applications on the Maude platform.

Our methodology focuses on testing run-time properties of Creol objects. By the very nature of Creol objects, of particular interest is to test run-time properties of the object state under different possible interleavings of its processes. In order to specify and execute such tests we need an appropriate abstraction of processes which focuses on their interleavings as described by the control structure of their release points. We do so by modeling the internal flow of control within a process between its release points into atomic blocks consisting of sequences of assignments. The release points of a process themselves then can be represented by the states of a finite automaton, also called a *method automaton* (because processes are generated by method calls). The transitions of a method automaton involve the assignments and a guard on the object state which specifies the enabling condition of the corresponding atomic block. We assume given a finite set of internal processes in an object, reflecting the message queue of incoming method calls for the object. The possible interleavings of this initially given finite set of processes is thus abstracted into the interleavings of their automata representations.

Scheduler automata further constrain the possible interleavings by means of abstract representations of the enabling conditions of the method automata. The automatically generated *scheduled system automaton* representing the possible interleavings of the method automata and the scheduler automaton is instrumented with test purposes, expressed as Boolean conditions over the method automata's state variables, that are attached to states.[5]

To compute test cases for a test purpose we search for paths that reach and fulfill the test purpose. We generate a set of such test cases by computing a test "harness" describing all paths in the model that will reach the test purpose. To this end, we use weakest precondition computation to propagate the conditions to the initial state of the system. The condition at the initial state describes the values that state variables can take for executing that test case, reflecting the actual parameters to the method calls in the message queue. Each possible path that reaches the condition(s) is its own test case.

The execution of a test on the Maude platform then checks whether the particular interleaving of the method automata described by the path in the system automaton can be realized by the Maude implementation of the Creol object such that it satisfies the conditions.

4 Combining Method Automata and Scheduling Policies

In this section, we present the symbolic transition system construction used to specify the system's behavior. We adapt the symbolic transition systems of [13], using shared variables for communication instead of input/output actions.

Syntax A Symbolic Transition System is a tuple $\langle Q, q_0, T, V \rangle$, where:

[5] Computing test cases that reach a certain condition in the program can be done with conditions that are simply *true*.

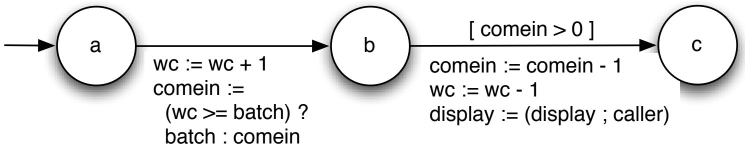

Fig. 3. Method Automaton of the `register()` method

- Q is a finite set of locations $q_i, i \geq 0$
- $q_0 \in Q$ is the initial location
- V is a set of variables
- T is a set of transitions of the form $\langle q, g, S, q' \rangle$, where
  - $q \in Q$ is the source location
  - g is a Boolean *guard expression* over V
  - S is a sequence of *assignment statements* changing the value of some $v \in V$
  - $q' \in Q$ is the target location

Semantics A *state* is a pair $\langle q, v \rangle$ consisting of a location q and a valuation v for the variables. For the initial state, $q = q_0$. Let *eval* be the function mapping an expression and a valuation to a result[6]. Then, for a state $\langle q, v \rangle$, *executing* a transition $\langle q, g, S, q' \rangle$ results in a new state $\langle q', v' \rangle$ where the new valuation v' is the result of evaluating all assignment statements in S, using *eval* with the former valuation v to calculate new values for the affected variables, provided that $\mathit{eval}(g, v) = \mathit{true}$.

4.1 Modeling Method Invocations: Method Automata

Invocations of methods on Creol objects are modeled by *Method Automata*, a slight extension of the symbolic transition systems described above.

A method automaton is a tuple $\langle m, Q_m, q_0^m, T_m, V_m, \mathit{Val}_m \rangle$ so that m is a unique identifier, Q is a set of locations q_i^m etc. Other than the systematic renaming of locations, the semantics are the same as for symbolic transition systems. Additionally, Val_m is a mapping $v \in V_m \mapsto x$ giving initial values x to all variables v. (Conceptually, Val_m models parameters passed to the method as well as initial values of local variables.)

A Creol method without release points is modeled as a method automaton with only beginning and end state. Each release point is modeled as an intermediate state where execution can switch to another running method.

By convention, the names of the local variables in a method automaton are prefixed with the unique identifier m of the automaton, so that the names are unique in the presence of multiple instances of the automaton. This approach

[6] In this paper, we use expressions over the integer and Boolean domains with the usual operations and semantics.

is sufficient since each invocation of a Creol method is modeled by its own automaton. Names of instance variables, such as `wc` and `display` in Fig. 3 are not prefixed in this way, since every method automaton has access to the same instance variables.

4.2 Modeling Parallelism: The System Automaton

A configuration of multiple method invocations running in parallel is modeled as a symbolic transition system as well. We shall refer to such an automaton as a *system automaton.*

Definition 1. *Let $A_i = \langle m_i, Q_{m_i}, q_0^{m_i}, T_{m_i}, V_{m_i}, Val_{m_i}\rangle$ be method automata (for $1 \leq i \leq n$). Define the composition of $A_1, \ldots, A_n$ as a system automaton $A = \langle Q, q_0, T, V, Val\rangle$ such that*

$$\begin{array}{l}
Q = \{\langle m_i, q^{m_1}, \ldots, q^{m_n}\rangle \mid \forall 0 < j \leq n : q^{m_j} \in Q_{m_j}\} \\
q_o = \langle m_1, q_0^{m_1}, \ldots, q_0^{m_n}\rangle \\
T = \left\{ \langle q, g, S, q'\rangle \left| \begin{array}{l} q \;= \langle m_l, q^{m_1}, \ldots, q^{m_i}, \ldots, q^{m_n}\rangle \wedge \\ q' = \langle m_i, q'^{m_1}, \ldots, q'^{m_i}, \ldots, q'^{m_n}\rangle \wedge \\ \langle q^{m_i}, g, S, q'^{m_i}\rangle \in T_{m_i} \wedge \forall j \neq i : q'^{m_j} = q^{m_j} \end{array} \right. \right\} \\
V = \bigcup_{0<i\leq n} V_{m_i} \\
Val = \bigcup_{0<i\leq n} Val_{m_i}
\end{array}$$

The semantics of executing a transition of the system automaton is that of executing the transition of *one* of the participating method automata ($q^{m_i} \leadsto q'^{m_i}$), leaving the state of all other method automata invariant ($q'^{m_j} = q^{m_j}$). Further note that the first element of the system automaton's state designates the method automaton which did the previous transition (for the initial state, it is arbitrarily set to m_1). Because of this, the transitions of the system automaton can be attributed back to a particular method automaton; this will become important in scheduling.

4.3 Modeling Schedulers: The Scheduler Automata

The system automaton as defined in Section 4.2 does not place restrictions on which method automaton executes at each step beyond the guards of the method automata transition themselves. We use a *scheduler automaton* to express additional restrictions on method automata execution in the system automaton.

A scheduler automaton is modeled as a labeled transition system. It is used to strengthen the guards on the transitions of a system automaton composed of method automata $m_1 \ldots m_n$, and hence, restrict which method(s) are allowed to run.

Definition 2. *Let A be a system automaton for methods $m_1, \ldots, m_n$. Define a scheduler for A as an automaton $S = \langle Q, q_0, T\rangle$ such that*

$$\begin{array}{l}
Q = \{m_i \mid 1 \leq i \leq n\} \\
q_0 = m_1 \\
T = \{\langle q, g, q'\rangle \mid q \in Q \wedge q' \in Q \wedge g \in G(A)\}
\end{array}$$

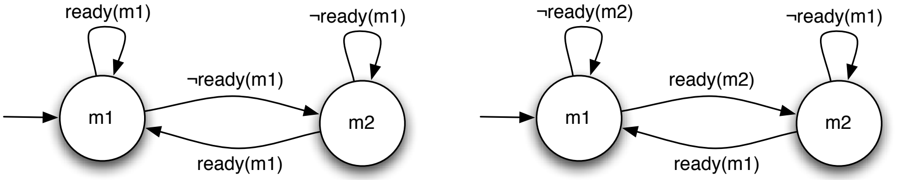

Fig. 4. Example scheduler automata: priority (left), round-robin (right)

The transitions on a scheduler automaton have guards $g \in G(A)$ in the form of *readiness predicates* that are defined in the following way: Given a system automaton A for methods $m_1, \ldots, m_n$, $G(A)$ is defined inductively by $\mathit{ready}(m_i) \in G(A)$ and $\neg \mathit{ready}(m_i) \in G(A)$ for $1 \leq i \leq n$, and $g_1 \wedge g_2 \in G(A)$ and $g_1 \vee g_2 \in G(A)$ if $g_1, g_2 \in G(A)$. The expression $\mathit{ready}(m_i)$ denotes a predicate which is *true* whenever the method automaton m_i has at least one *enabled transition* (i.e., whose guard evaluates to *true*) in the current state of A.

The scheduler automaton has n states, one for each method automaton in the system automaton. Each scheduler state is labeled with one method automaton's unique identifier m_i. The label on the current state of the scheduler automaton names the method automaton that executed the most recent transition of the system automaton. By definition, m_1 is the scheduler automaton's initial state.

Figure 4 shows two scheduling automata, both for a system automaton with two method automata m_1 and m_2: a simple priority scheduler that always gives preference to m_1 over m_2, and a round-robin scheduler.

4.4 Integration of the Scheduler and the System Automaton

The scheduling of tasks in a system automaton according to the policy expressed by a specific scheduler automaton is done in the following way:

For each state $q = \langle m_k, \ldots \rangle$ of the system automaton, find the corresponding state m_k of the scheduler automaton. For each transition $t = \langle q, g, S, q_1 \rangle$ in the system automaton, take the scheduler automaton's transition that enables t, i.e. the transition that leads to the scheduler state m_i if $q_1 = \langle m_i, \ldots \rangle$. If there is no such scheduler transition, remove the transition from the system automaton (since the scheduler does not allow the method automaton m_i to run after m_k). Otherwise, strengthen the guard on the transition t by the guard expression on the scheduler transition from m_k and m_i, replacing all sub-expressions $\mathit{ready}(m_x)$ with the disjunction of the guards on all transitions of method automaton m_x in its current state.

We refer to a system automaton which is scheduled by a scheduler automaton as a *scheduled system automaton.* Formally, we define the expansion of readiness predicates for specific states of a system automaton and a scheduled system automaton as follows.

Definition 3. *Let $A = \langle Q, q_0, T, V, \mathit{Val} \rangle$ be a system automaton for the methods $m_1, \ldots, m_n$. For a state $q \in Q$ and a scheduler guard $g \in G(A)$, scheduler guard*

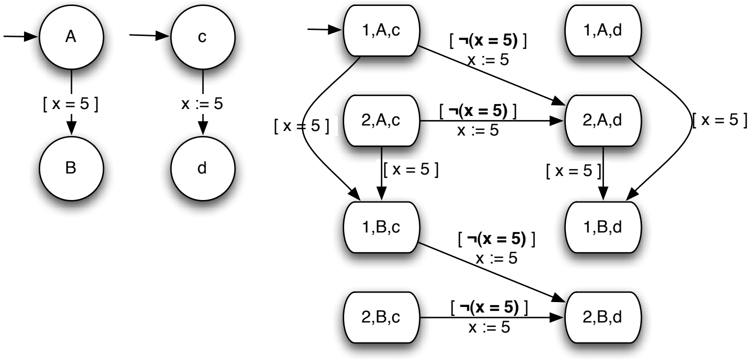

Fig. 5. Two simple method automata and a system automaton consisting of the two automata running in parallel under the priority scheduler of Figure 4 (guards in bold added by the scheduler).

expansion is a function $[\![g]\!]_q$*, inductively defined as follows:*

$$\begin{aligned}
&[\![ready(m_i)]\!]_q = \bigvee\{g \mid \langle q, g, S, q_1\rangle \in T \wedge q_1 = \langle m_i, q^{m_1}, \ldots, q^{m_n}\rangle\}\\
&[\![\neg ready(m_i)]\!]_q = \neg[\![ready(m_i)]\!]_q\\
&[\![g_1 \vee g_2]\!]_q = [\![g_1]\!]_q \vee [\![g_2]\!]_q\\
&[\![g_1 \wedge g_2]\!]_q = [\![g_1]\!]_q \wedge [\![g_2]\!]_q
\end{aligned}$$

In the first part of Definition 3, we use the disjunction on a set to denote the disjunction of all the elements in the set.

Definition 4. *Let* $A = \langle Q_A, q_0^A, T_A, V_A\, Val_A\rangle$ *be a system automaton for methods* $m_1, \ldots, m_n$ *and let* $S = \langle Q_S, q_0^S, T_S\rangle$ *be a scheduler. Define a scheduled system as an automaton* $SA = \langle Q, q_0, T, V, Val\rangle$ *such that*

$$\begin{aligned}
&Q = Q_A\\
&q_0 = q_0^A\\
&T = \left\{ \langle q, g, S, q'\rangle \;\middle|\; \begin{array}{r} q = \langle m_l, q^{m_1}, \ldots, q^{m_n}\rangle \wedge q' = \langle m_i, q'^{m_1}, \ldots, q'^{m_n}\rangle \\ \wedge\langle q, g', S, q'\rangle \in T_A \wedge \langle m_l, g'', m_i\rangle \in T_S \wedge g = (g' \wedge [\![g'']\!]_q) \end{array} \right\}\\
&V = V_A\\
&Val = Val_A
\end{aligned}$$

For example, if the transition guard on the scheduler is $[\neg ready(m)]$ and automaton m in its current state has two transitions with the guards $[x <= 5]$ and $[x > 5]$, then relevant guards on the transitions in the system automaton will be strengthened with $\neg(x <= 5 \vee x > 5)$. Transitions whose guards reduce to *false* (as in this example) can be eliminated from the system automaton.

5 Test Case Generation with WP and Schedulers

We use a scheduled system automaton *SA* (see Definition 4) to test the Creol object it represents. *SA* contains all runs an object can perform for a given

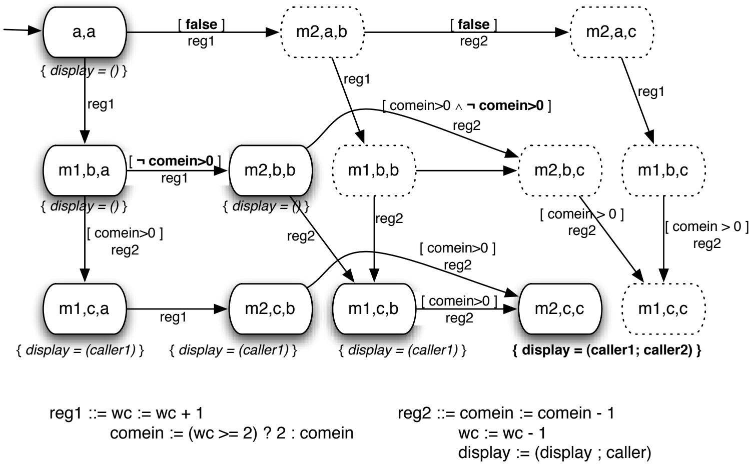

Fig. 6. A scheduled system automaton with two method automata for the `register` method, under priority scheduling and with batch size 2. Guard terms in bold are added by the scheduler, states that are unreachable under priority scheduling are dashed.

initial message queue and scheduler. In the following, we give an approach to computing test cases of interest from this automaton.

Specifically, we define how to compute the weakest precondition (WP) for a scheduled system automaton and use this technique to generate test cases according to a *test purpose*.

The intention of the test cases to generate is captured by *test purposes*, which are abstract specifications of actual test cases. In conformance testing, the notion of a test purpose has been standardized [7]:

Definition 5 (Test purpose, general). *A description of a precise goal of the test case, in terms of exercising a particular execution path or verifying the compliance with a specific requirement.*

In our setting, these requirements are expressed by assert statements in a system automaton. The condition p of an assert has to be fulfilled in all possible runs leading to the assert. (For simplicity, we will use p to refer to the assertion and its condition synonymously.) To compute test cases for a test purpose, we search for paths that reach and comply with all its assert statements. Intuitively, this corresponds to computing the *weakest precondition* for p. In the following we will, without loss of generality, concentrate on test purposes that can be specified with a single assertion. Conditions for the general case are computed by combining the results from the single conditions.

Figure 6 shows the graph of a system automaton that models two invocations of the `register` method and `batch` size `2`, scheduled with the priority scheduler from Figure 4. This scheduler removes the edge from the initial state (a,a) to (m2,a,b) because both processes are enabled (with m1 having priority).

Consequently, a portion of the state space of the system automaton becomes unreachable in the scheduled system automaton and can be removed.

Figure 6 also shows the additional conditions from scheduling on the edges. E.g., in state (m1,b,a) process m2 is only enabled if *comein* is not > 0. The test purpose is to compute test cases to reach state (m2,c,c) with $\mathit{display} = (\mathit{caller1}; \mathit{caller2})$. We constrain ourselves to only illustrate the WP computation for the *display* variable, whose computed value is depicted in curly brackets. Computing the WP to the initial state results in an empty *display* variable, for which all paths reach the desired state[7]. The actual implementation must not block for this input and must satisfy the assertion.

To test the intermediate and final assertions on the Creol model, we create a *test harness* H. The harness is constructed from the system automaton A as $H = \langle Q_A, q_0^A, T_A, V_A, c(Q_A)\rangle$, with Q_A, q_0^A, T_A and V_A reflecting the system automaton, and $c(Q_A)$ a condition defined for each location of A, representing those valuations in a location that only occur in runs that eventually will reach and comply with p. Thus, for every valuation in $c(Q_A)$ two properties hold: (1) there is a transition such that the destination is again in $c(Q_A)$ and (for determinism) (2) there is no transition such that the destination is not in $c(Q_A)$. Using standard weakest-precondition predicate transformers *wp* for our simple statements S (assignments and sequential composition only), we have:

$$c_p(q) = \bigvee_{\forall \langle q,g,S,q'\rangle \in T} wp\,(S, c(q')) \wedge g \tag{1}$$

$$c_{\neg p}(q) = \bigvee_{\forall \langle q,g,S,q'\rangle \in T} wp\,(S, \neg c(q')) \wedge g \tag{2}$$

$$c(q) = c_p(q) \wedge \neg c_{\neg p}(q) \tag{3}$$

We compute $c(Q_A)$ iteratively by setting $c_0(q) = p$ for $q = q_p$ and $c_0(q) = \mathit{false}$ for all other locations. The first iteration will result in all states that reach p in one step, then those with distance two and so forth. The iteration steps are sound: each iteration results in valuations that give valid test cases. This is an important observation because although this process always results in a fixed point for finite state systems (cf. CTL model checking of $\mathbf{AF}\,p$ [2]), the state space for STS is infinite and the iteration might not terminate. Soundness allows us to stop computation after a certain bound or amount of time even if no fixed point is reached yet. Any initial state in $c(q_0)$ gives valid test cases even if no fixed point can be computed.

The test case for the scenario of Figure 6 consists of the following:

- A list of method invocations $(\langle m_1,$`register()`$\rangle, \langle m_2,$`register()`$\rangle)$
- The priority scheduler from Figure 4
- The initial value `()` for the instance variable `display`
- The test harness H, giving verdicts at each scheduling decision point

[7] The representation is strongly simplified, exact computation will give more conditions on the states and unveils that only the path using the edge (m1,b,a)(m2,b,b) is feasible.

5.1 Test Case Execution

The test driver in Creol uses the scheduler to guide the Creol model and the test harness H to arrive at test verdicts. The initial values and method parameters are chosen such that condition $c(q_0)$ is fulfilled, at each release point of the Creol object, the conditions on the harness are checked. At each release point, the scheduler chooses among the enabled processes to continue the execution. There are two different ways of arriving at a test verdict of *Fail*:

- If the Creol object does not fulfill the current condition of the harness, the implementation of the last executed basic block violates the specification by the method automaton.
- If the condition is fulfilled but no process is enabled (the test process deadlocks), the implementation fails to handle all the valuations that are required by the model.

If the test harness arrives at the terminating state and the condition is fulfilled, a test verdict of *Success* is reached.

Strengthening the Guards of the Harness. The computation as shown above uses the weakest precondition to reach the test purpose p, or, in other words, the set of initial states that reach the test purpose in every legal run. Input values that might miss p due to non-determinism are ignored. To achieve optimal test coverage, however, it is desirable to search for all input values that *can* fulfill the test purpose and add enough information to H for the test driver to guide the run to the desired state. In other words, instead of computing those initial states that will reach p in every run, we want to compute states for which a run *exists*.

The annotated automaton provides us with a simple mechanism to achieve this goal. For the necessary adjustments we have a second look at the computation of $c(Q_A)$. Formula (1) represents the states that can reach p, while those states that can avoid p are removed using Formula (2). If we don't consider $c_{\neg p}$ in Formula (3), we compute all valuations for which a run to p *exists*, but the test driver has to perform the run on a trial an error basis: executing a statement and checking if the result still can reach p, backtracking otherwise. To avoid this overhead, we add new guards g' to H to restrict the runs to those valuations that always can reach p:

$$g'(< q, g, S, q' >) = g \wedge wp(S, c(q'))$$

Using g' for the computation of $c(Q_A)$ results in all states for which a run to p exists, which easily can be seen by inserting g with $g \wedge wp(S, c(q'))$ in formulae (1) and (2):

$$c'_p(q) = \bigvee_{\forall \langle q,g,S,q' \rangle \in T} wp\left(S, c(q')\right) \wedge g \wedge wp\left(S, c(q')\right) = c_p(q)$$

$$
\begin{aligned}
c'_{\neg p}(q) &= \bigvee_{\forall \langle q,g,S,q'\rangle \in T} wp\,(S, \neg c(q')) \wedge g \wedge wp\,(S, c(q')) = \mathit{false} \\
c'(q) &= c'_p(q)
\end{aligned}
$$

Using g' as guards for the test driver excludes all transitions to states that cannot reach p. This allows to avoid unnecessary backtracking while examining all paths that can be extended to reach the test purpose, resulting in a larger variety of possible runs and better coverage. The approach does not come without obstacles though, g' only points to states that *can* reach p — the test driver needs to be able to detect loops to make sure to finally reach it. Furthermore, a path to p might not be available in the implementation. If the only available path avoids p, the test driver has to backtrack to find a path to p.

6 Related Work

With the growing dependency on distributed systems and the arrival of multicore computers, concurrent object-oriented programs form a research topic of increasing importance. Automata-based approaches have previously been used to model concurrent object-oriented systems; for example, Kramer and Magee's FSP [10] use automata to represent both threads and objects, abstracting from specific synchronization mechanisms. However, they do not address the issue of representing specific scheduling policies that we consider in this paper. Schönborn and Kyas [14] use Streett Automata to model fair scheduling policies of external events, with controlled scheduler suspension for configurations that deadlock the scheduler.

A lot of work is done in the area of schedulability which mainly deals with the question if a scheduler exists which is able to meet certain timing constraints (e.g., [12, 6]), but does not look into the functional changes imposed by different application-level scheduling policies. Established methods for testing object-oriented programs like unit-testing, on the other hand, deal with the functionality on a fine grained level, but fail to check for the effects of different schedulers (see e.g., [18]). Instead, the main challenge for testing concurrent programs is to show that the properties of interest hold independent of the used scheduler. In contrast, the approach we have taken in this paper is to test properties of a program under a specific scheduling regime.

Stone [15] was the first proposing the manipulation of the schedules to isolate failure causes in concurrent programs. Her idea was to reduce the non-determinism due to scheduling by inserting additional break points at which a process waits for an event of another process. In Creol, this could be achieved by inserting additional `await`-statements. However, dealing with a modeling language, we prefer the more explicit restriction of non-determinism by modeling the scheduling policy directly. More recently, Edelstein et al. [5] manipulated the scheduler in order to gain higher test coverage of concurrent Java programs. They randomly seeded *sleep*, *yield* or *priority* statements at selected points in order to alter the scheduling during testing. This approach is based on the observation that a given scheduler behaves largely deterministic under constant operating conditions; by running existing tests under other scheduling strategies, additional timing-related errors are uncovered. Choi and Zeller [1] change schedules of a program to show the cause of a problem for a failing test case. They use DEJAVU, a capture/replay tool that records the thread schedule and allows the replay of a concurrent Java program in a deterministic way. Delta-debugging is used to systematically narrow down the difference between a passing and failing thread schedule. This approach helps in order to check if programs work under different schedules, but unlike the method shown in this paper do not help in the actual generation of the test case.

Jasper et al. [8] use weakest precondition computation to generate test cases especially tailored for a complex coverage criterion in single threaded ADA programs. Rather than augmenting the model, they generate axioms describing the program and use a theorem prover to compute its feasibility. More recently, [17] use weakest precondition to identify cause-effect chains in failing test cases to localize statements responsible for the error (fault localization). WP computation is furthermore used in several abstraction algorithms to identify relevant predicates for removing infeasible paths in abstract models. In [16], Tillmann and Schulte introduce "parametrized unit tests", which serve as specifications for object oriented programs. They use symbolic execution to generate the input values for the actual test cases. However, none of these approaches use WP computation for test case generation in concurrent systems.

7 Conclusion and Future Work

This paper presents an approach to generating test cases for concurrent, object-oriented programs with application-specific schedulers. The scheduling policy becomes part of the test case in order to control its execution. We therefore introduce an automaton approach for specifying the behavior of both the system and the scheduler, as well as its composition and extension to a harness for a test driver. Enforcing a scheduling regime limits the non-deterministic interleavings of behavior, a well-known problem in testing and debugging of concurrent systems. A further important aspect is that the separation of concerns between functionality and scheduling allows scheduling issues, which are crucial in concurrent programs, to be specified and tested at the abstraction level of the executable modeling language.

In this paper, we expect the method automata and scheduler to be given as specifications, and check for compliance with a given Creol implementation. A natural extension for future work is to automatically construct the method automata from the Creol code and check against different schedulers for compliance. The test driver will be implemented within the Maude interpreter for Creol, which allows the test driver to influence the scheduling.

Further future work comprises the extension to schedulers with internal state to express more involved scheduling strategies and to extend our approach with further features of object-oriented languages.

References

1. J.-D. Choi and A. Zeller. Isolating failure-inducing thread schedules. In *International Symposium on Software Testing and Analysis*, pages 210–220. ACM, 2002.
2. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* The MIT Press, Cambridge, Massachusetts, 1999.
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, Y. Nir, G. Ratzaby, and S. Ur. Multithreaded Java program test generation. *IBM Systems Journal*, 41(1):111–125, February 2002.
6. E. Fersman, P. Krcál, P. Pettersson, and W. Yi. Task automata: Schedulability, decidability and undecidability. *Information and Computation*, 205(8):1149–1172, 2007.
7. ISO/IEC 9646-1: Information technology - OSI - Conformance testing methodology and framework - Part 1: General Concepts, 1994.
8. R. Jasper, M. Brennan, K. Williamson, B. Currier, and D. Zimmerman. Test data generation and feasible path analysis. In *Proceedings of the International symposium on Software testing and analysis (ISSTA'94)*, pages 95–107. ACM, 1994.
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. J. Magee and J. Kramer. *Concurrency: State Models & Java Programs.* Wiley, 2 edition, 2006.
11. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.
12. L. Nigro and F. Pupo. Schedulability analysis of real time actor systems using coloured petri nets. In *Concurrent Object-Oriented Programming and Petri Nets: Advances in Petri Nets*, volume 2001 of *LNCS*, pages 493–513. Springer, 2001.
13. V. Rusu, L. du Bousquet, and T. Jéron. An approach to symbolic test generation. In *Proceedings of the 2nd International Conference on Integrated Formal Methods (IFM'00)*, volume 1945 of *LNCS*, pages 338–357. Springer, 2000.
14. J. Schönborn and M. Kyas. A theory of bounded fair scheduling. In J. Fitzgerald and A. Haxthausen, editors, *International Colloquium on Theoretical Aspects of Computing (ICTAC)*, LNCS. Springer, 2008. (This volume).
15. J. M. Stone. Debugging concurrent processes: A case study. In *Proceedings SIGPLAN Conference on Programming Language Design and Implementation (PLDI'88)*, pages 145–153. ACM, June 1988.
16. N. Tillmann and W. Schulte. Parameterized unit tests. In *Proceedings of the 10th European Software Engineering Conference / 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'05)*, pages 253–262. ACM, 2005.
17. C. Wang, Z. Yang, F. Ivancic, and A. Gupta. Whodunit? Causal analysis for counterexamples. In *International Symposium on Automated Technology for Verification and Analysis (ATVA'06)*, volume 4218 of *LNCS*, pages 82–95. Springer, 2006.
18. E. J. Weyuker. Testing component-based software: A cautionary tale. *IEEE Software*, pages 54–59, September 1998.