Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 243 (2009) 3–14

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Modeling and Testing Multi-Threaded Asynchronous Systems with Creol[⋆]

Bernhard Aichernig[a,b,1] Andreas Griesmayer[b,2]
Rudolf Schlatte[a,b,3] Andries Stam[c,4]

[a] *TU Graz, Austria*

[b] *UNU-IIST, Macao*

[c] *Almende BV, The Netherlands*

Abstract

Modeling concurrent systems and testing multi-threaded implementations against the model is an exciting field of study. This paper presents work done on constructing and executing test cases for an industrial-size multi-threaded application against a model written in the Creol modeling language. Models written in Creol, an object-oriented, concurrent modeling language, can be structurally similar to the finished implementation; we show how to keep this desirable property when re-using Creol models as test oracles. Also, a conformance relation between model and system under test that needs less controllability than other relations that are based on automata is presented.

Keywords: Software Testing, Modeling, Parallelism

1 Introduction

Formal testing of single-threaded programs can rely on a rich body of theory and industrial experience [11,19,10]. Formal testing of multi-threaded or distributed systems, on the other hand, is still an open area of research. This paper presents work on modeling a concurrent system and testing the system against the model.

As modeling language, Creol [13] is used. Creol is an object-oriented, distributed modeling language that has, in our experience, proved capable of modeling the behavior of large parallel software systems. Because of Creol's expressiveness, the

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

1 Email: aichernig@ist.tugraz.at

2 Email: agriesma@iist.unu.edu

3 Email: rschlatte@iist.unu.edu

4 Email: andries@almende.org

doi:10.1016/j.entcs.2009.07.002

models can have similar code structure to the implementation (e.g. with respect to method names and flow of control); this helps modeler and implementer to have a common understanding and vocabulary.

One main contribution of this paper is to show a way to instrument existing Creol models so they can be used for testing purposes, without having to restructure or rewrite the models. Another contribution of the paper is to present a conformance relation between a model and an implementation in the face of minimal controllability of both implementation and (operational semantics of) model.

We verify our testing approach through a case study based on ASK, an industrial software system for connecting people to each other via a context-aware response system. A substantial part of the ASK implementation, which is mainly written in C, has been modeled in Creol. The ASK system is inherently multi-threaded and uses asynchronous communication.

The rest of the paper is organized as follows: Section 2 presents some related work in the area of formal testing, followed by a presentation of the Creol modeling language in Section 3. Section 4 presents the ASK system, the case study that we use in Section 5 to present our approach in detail. Finally, Section 6 gives conclusions and discusses possible future work.

2 Related Work

There is considerable previous work on the use of formal methods for testing components [11,17]. Various conformance relations have been proposed, with varying demands w.r.t. controllability and observability placed on the *system under test* (SuT). As an example, the *ioco* conformance relation is widely used in the literature, as well as in available testing tools like TGV [10], TestGen [12] and TorX [4].

ioco stands for *input/output conformance* and requires that during a test run, inputs to the SuT are selected by the tester while outputs are observed by the tester. After each run that is allowed in both the specification and the SuT, every output of the SuT has to be possible in the specification. While this conformance definition (and some derivations of it like in [19]) is useful for many applications, it requires that SuT and tester can be synchronized, i.e. that after some sequence of output actions, the implementation waits for an input action from the tester.

In our application, however, the components are coupled asynchronously. Input *actions* emitted from the environment are put in a queue. They are processed in any order determined by the implementation, emitting input *events*. A test verdict is reached by observing the input events interleaved with output events.

Asynchronous I/O is studied in [15] by introduction of *queued testing*. The test process is decomposed into subprocesses to produce input and output sequences according to a test case. This approach yields a weaker conformance relation than *ioco*, because it does not capture relations (cause-effect-chains) between input and output; on the other hand, this approach places fewer controllability demands on the implementation (in original *ioco*, the tester is not input enabled, hence might not be prepared to accept output from the SuT, although this has been revised in [18]). We expand upon that work by dropping the need to distinguish between input and output while monitoring events, thus 1) capturing relations between input and outputs and 2) allowing to monitor events that can be stimulated both from the tester as well as the SuT itself.

The idea of modeling languages with operational semantics that can be used for testing is not new. A recent example is Microsoft's Spec Explorer [5], which models observable and controllable events ("Actions" in their parlance) as methods, with preconditions that tell when events can occur. Test cases are constructed by calculating a state machine and then generating traces of events, replaying them on the SuT. The big advantage of that model is the automated test case generation; in our approach the initial configuration of events must be authored manually. On the other hand, the models in Spec Explorer are geared towards testing, and observation of events is always on the method call level. In our approach, the models can be written in a style that might be more familiar to programmers and more useful for initial system modeling. The same models can then be re-used for testing with minimal effort.

3 The Creol Modeling Language

This paper describes the use of executable Creol [13] models for testing concurrent systems. The operational semantics of the Creol language is defined formally in rewriting logic [14] and is executable on the Maude platform [6].

Creol is an object-oriented, distributed modeling language. Objects can be *active* (having a dedicated method that is started upon object creation) or passive (only reacting to messages).

In contrast to, e.g., Java, each Creol object encapsulates its state and external manipulations only can happen through calls to the object's methods. A method invocation corresponds to the creation of a process to handle that invocation. Calls to methods are asynchronous, that is, a method call and its return are separate events. Each object has its own processor and process queue, and method calls do not block the caller but are simply inserted into the callee's process queue. The order in which two consecutive calls are executed can not be influenced by the caller. A method call is represented in the caller as a *Future Variable* [8], which gets a value once the call has finished and returned. The caller can wait on the future variable, but may choose to do some work between calling a method and collecting the results of the call.

Each object has its own object-internal (and not further specified) scheduler. In a standard setting, there are no assumptions that can be made about the order of process execution. (For an approach to add schedulers to Creol objects, see [16]) Only one process per object is active at a time. Creol processes do not use preemption. Instead, explicit conditional suspension points (in form of `await` statements) are used to release a process and allow another process to execute. Cooperative scheduling might be inefficient for a production language, but has great benefits for modeling. Because scheduling points are explicit, race conditions can often be found by visual code inspection. Hence, it is very easy to achieve thread-safety for Creol objects.[5]

The aforementioned properties of Creol make the language well suited for modeling distributed systems. Typically, an object represents one module of the system, where modules communicate through clearly defined interfaces. Methods are annotated with a *co-interface*, allowing both the restriction of the possible callers to the method and a way of call-back using the **`caller`** variable. The **`prove`** statement and invariants on methods allow for checking conditions during runtime.

3.1 Creol Syntax

We give a short overview of the Creol syntax used in this paper. Method definitions are of the form:

```
op method_name (in arguments; out return_values) == statement_list
```

where the identifiers named in `return_values` can be used like local variables. The values bound to these identifiers are passed back to the caller when the method finishes, via the **`return`** statement or by reaching the end of `statement_list`.

The **`await`** statement releases the process if its condition is not satisfied. In that case, the process is put into the object's process queue and can be rescheduled when the condition becomes **`true`**.

A method call has the form

```
l!obj.method(arguments)
```

and the collection of the methods return values has the form

```
l?(return_values)
```

where `l` is the label (future variable) of the call. The `l?` statement to wait for the return values is *blocking*, which means the process is not released and no process in the caller is executed until the called method returns. To release the process while waiting for a method to finish, the statement **`await`** `l?` is used. A blocking method call can be implemented by calling the method and immediately waiting for its return. The code in this paper mainly uses the syntactic form

```
obj.method(arguments ; return_values)
```

that combines method call and return value collection to implement blocking calls.

4 Case Study: The ASK System

ASK is an industrial software system for connecting people to other people via a context-aware response system. ASK has been developed by Almende [1], a Dutch research company focusing on the application of self-organisation techniques in human organisations and agent-oriented software systems. The system is marketed by ASK Community Systems [2]. ASK provides mechanisms for matching users requiring information or services with potential suppliers. Moreover, it is often used as a planning and scheduling system for the recruitment of skilled workers for various

[5] Implementing race conditions is possible but takes explicit effort: to investigate the cause of a suspected bug, we introduced an extra object in the model for a global variable in the implementation.

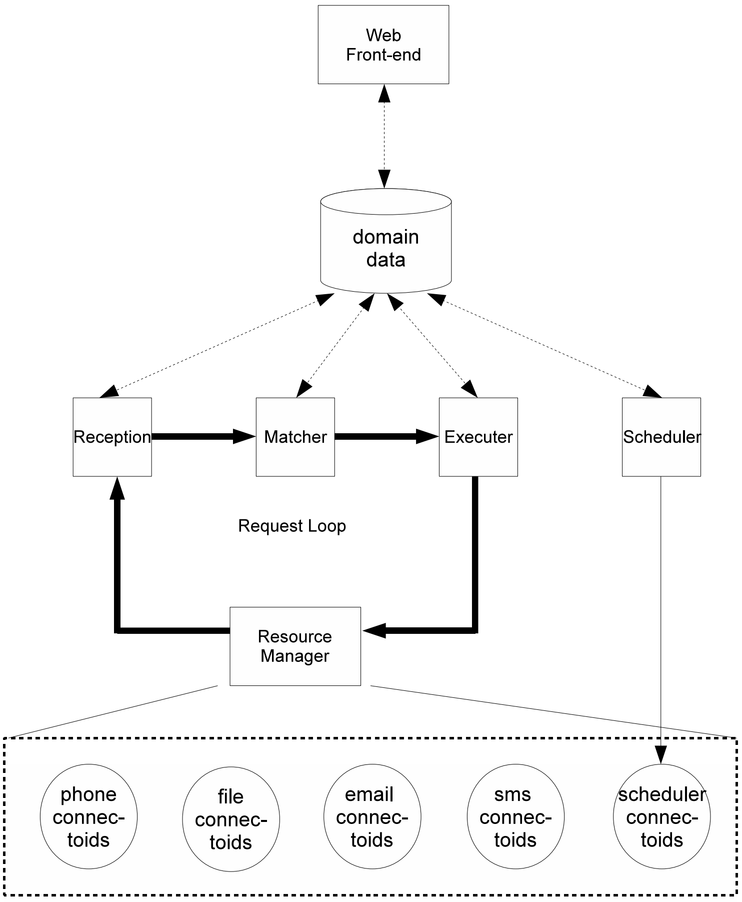

Fig. 1. ASK system architecture (simplified)

situations. Typical applications for ASK are workforce planning, customer service, knowledge sharing, social care and emergency response. Customers of ASK include the originally Dutch mail distribution company TNT Post and the cooperative financial services provider Rabobank. The amount of people connected and involved in an ASK system configuration may vary from several hundreds to several thousands.

4.1 High-level Overview

Figure 1 shows a simplified architectural view of the existing ASK system. The "heartbeat" of the system is the *Request loop*, indicated with thick arrows. A request contains, among other things, the information of two *participants* (a requester and a responder). The *Reception* component determines, based on information in the initial request, which actions are needed to fulfil the request. The *Matcher* component, if needed, searches for appropriate participants for the request. The *Executer* determines the best way in which the participants can be connected, or the best way in which a request can be fulfilled. The *Resource Manager* component effectuates the creation, deletion and reconnection of so-called *connectoids*, which represent specific media currently in use (a connected phone call, a played sound file, an email being written, an SMS message to be sent). The *Scheduler* component, finally, schedules certain requests based on job descriptions inside the database, like the request to recruit a certain amount of people for a certain job.

As an example, consider the request from a user of the ASK system to get in contact with a service supplier. Once the user contacts the ASK system, a connectoid created by the resource manager represents the incoming open call. The new connectoid is used to create an initially almost empty request, containing only the calling number and the number that was called (ASK systems typically support multiple call-in numbers). The request is sent to the reception, which e.g. presents an interactive voice response menu to the user. This could involve the playing of sound files (the menu choices being presented). In that case the request iterates through the components, in the meanwhile causing the creation of sound file connectoids in the resource manager, which are connected to the open call-in line to the user (the user listens to the sound file). As soon as the actual request is clear, namely to get in touch with a service supplier, the matcher searches for the appropriate supplier registered in the database. The matching can be based on various sources of information, including feedback from users about the quality of the supplier and its current reacheability (based on a time table in the database). In the end, the resource manager sets up a connection to the service supplier, via e.g. a phone connectoid. After that, the scenario continues with e.g. dial tones entered by both participants, hangup of one participant, or even hangup of both, in which case the request could cease to exist.

4.2 Implementation Details

The ASK components have been implemented in C. Each component is itself multi-threaded. The threads act as workers in a thread-pool, which execute tasks put into a component-wide shared task queue. Within a single component, threads do not communicate with each other directly. However, they can dispatch new tasks in the task queue, which are then eventually executed by another or the same thread. Threads are also able to send messages to other components. Incoming messages for a component are received by a thread executing a special *hostess* task, which continually checks for incoming messages and dispatches tasks accordingly. In most of the components, the amount of threads can change over time, depending on the amount of pending tasks in the task queue and the amount of idle threads.

4.3 Case Study Scenario

The scenario that we use as an example throughout this paper models the creation, dispatch and execution of tasks inside the ASK components. However, this introduction to the ASK system already shows that more complex scenarios are thinkable, e.g. by checking request flows. We expect to report on more elaborate testing scenarios in the near future.

For now, we consider the following scenario: The scheduler component, which reads its jobs from the ASK database, is provided with a database containing a single job. This job, once retrieved and executed by the scheduler, results in the creation of many tasks, inside the scheduler itself but also certainly inside other components, like the reception. This depends heavily on the precise job for the scheduler, as configured in the database. For example, issuing the recruitment of ten individuals for a certain service could cause the creation of ten callout requests to the reception component, resulting in ten new tasks inside that component. However, if some people have recently been called for recruitment, or no contact information of people can be found in the database, the amount of callout requests could be smaller.

As a test scenario, we consider the verification of the correct dispatching of tasks based on the contents of the scheduler job and the database.

5 The Test Process

In testing, we initiate a run of the SuT (System under Test) and check if the resulting run behaves as expected. For synchronous systems, this can be done by building a *test graph*, which relates inputs given to the SuT with the outputs returned from the SuT. Depending on the outputs, new inputs can be selected to reach a certain goal in the test graph.

In the setting of asynchronous, concurrent systems, however, this is not possible. In general, the system does not "wait" for the tester to send inputs if there are still open tasks to perform. Waiting with sending inputs until the system finished all open tasks is a bad option because that would eliminate important test scenarios. Furthermore, a pre-computed test graph would be enormous due to the possible interleaving inherent to concurrent systems. Instead, we use the Creol model as oracle for the test run. To test if an execution of the SuT is valid, the tester tries to execute it in the model too – only if that is possible, the run is valid.

5.1 Test Methodology

In the following we assume that we have a Creol model that completely models and has similar structure to the SuT. Due to their similar structure, both model and SuT can be annotated with the same events (modulo an abstraction function π) (see Subsection 5.3 for a discussion of events and instrumentation). The principle of the approach is quite simple: If the SuT is correct, then for each initial configuration and sequence of observable events in the implementation, a tester shall be able to observe the same sequence of events (lifted into the model domain) in the model.

Formally, the implementation can be seen as a function I from an initial configuration to a set of event traces: $conf_I \xrightarrow{I} \{events_I\}$. Similarly, the operational semantics of the model maps an initial configuration $conf_M$ to a set of event traces $events_M$. Each element of $\{events_I\}$ resp. $\{events_M\}$ represents a possible sequence of observable events in response to the initial configuration in implementation or model. The results of both M and I are sets because of the nondeterministic nature of process scheduling.

An abstraction function π projects configurations and event traces from implementation to model. This results in the following diagram:

(1)
$$\begin{tikzcd} conf_M \arrow[r, "M"] & \{events_M\} \\ conf_I \arrow[r, "I"] \arrow[u, "\pi"'] & \{events_I\} \arrow[u, "\pi"'] \end{tikzcd}$$

If the model and implementation have exactly the same observable behavior regarding their event traces, this diagram commutes. But this is not necessary for the implementation to conform to the model – an implementation behaves according to a model if the following holds for all initial configurations $conf_I$:

(2)
$$\pi(I(conf_I)) \subseteq M(\pi(conf_I))$$

Informally, this conformance relation says that the projection of all possible sequences of events observable in the implementation must be contained in the set of sequences of events observable in the model. The objective of testing is to try to find a counter-example for the above relation – to find a scenario where I exhibits behavior not covered by M.

In order to be able to verify the conformance relation, we introduce a *tester*, a process actor who supplies the initial configuration to the model and restricts the order of observable events during execution of the model. We restrict process scheduling in the model at carefully chosen points so that the model can only proceed past an observable event when the tester, who knows the sequence of events recorded from the implementation, allows it. The rest of this section describes the implementation of this approach in detail.

5.2 Actions and Events

Our test assumption is that a sequence of events that is observed on the implementation can be reproduced (replayed) by the model. Usually in the testing literature, both implementation and model are specified as some variant of Input/Output Labeled Transition Systems (IOLTS). In that model, events are separated into *Input* and *Output Actions* that occur interleaved; this is the basis of **ioco** [17] and indeed much of the formal testing theory.

In our case, the situation is slightly different. Like in [15], input and output can be performed independent from each other. Consequently, we distinguish between (controllable) *actions* and (observable) *events*.

An action is a stimulation to SuT and model, while an event testifies that something happened in the system. E.g., a method call from the tester is an action, the start of execution of that method is a related event. Because of the asynchronism of our systems, several events might occur between a method call (the action) and its execution (the event). Likewise, the order in which methods are executed might be different from the order of the calls.

An action is always initiated by the tester. Some events (like, say, the start of execution of a method `create_task`) are the direct consequence of actions (a call to a method `create_task`). The same events can potentially be observed in the

```
op dispatchTask(out index: Int) ==
  await tc.request("dispatchTask"); // <-- the added call
  await openCounter > 0;
  index := index(states, "OPEN");
  states := replace(states, "BUSY", index);
  openCounter := openCounter - 1
```

Fig. 2. The dispatchTask method of the TaskQueue class of the ASK system. The first line of the method body signals the event dispatchTask to the testcase tc and requests permission to continue.

SuT without being the direct consequence of an action by the tester as well. In order to increase testability, event probes in the implementation should be placed such that they reflect when an action is accepted in the SuT.

5.3 *Adding Instrumentation*

As mentioned above, the language Creol is expressive enough that model and implementation can have a similar structure with respect to function/method names and control flow. Consequently, SuT and model can be instrumented to produce equivalent events. This subsection describes the technicalities of producing events.

5.3.1 *Instrumenting the Implementation*

We use Aspect-C [3] to insert event recording points into the existing code for the ASK system. Actions (incoming phone calls and emails, tasks to be started) are created by a test driver that runs in parallel with the ASK system. Typically, the following events are recorded:

- Task read from database, task added to queue, task claimed by worker thread.
- Outgoing phone call, incoming phone call, key pressed on phone, phone hangup.
- worker thread created, worker thread removed.

Other events can be added depending on the needs of the test case.

5.3.2 *Instrumenting the Model*

While the instrumentation in the SuT merely emits the events, the code of the model is changed such that the tester is able to steer it to verify the sequence of events performed by the SuT (see Section 5.1 for the theoretical basis of this approach). So, at the time when an event occurs in a model, the tester can delay or entirely disallow (infinitely delay) the process that signals the event.

For each event, a *Counting Semaphore* is used to synchronize the model and the tester. For each event, a `request` call is inserted at the point where the event occurs:

```
await tc.request("eventX");
```

Figure 2 shows the `dispatchTask` method of the TaskQueue class of the ASK system; in the model of the ASK system, this method is called by the worker threads to remove a task from the queue.

Configuration	Events	Tester

createTask	createTask
	↓
	dispatchTask
	↓
	createTask
	↓
	dispatchTask
		`op run ==`
		`  var t: Label[ ];`
		
		`  // The initial configuration`
		`  t!queue.createTask(taskId);`
		
		`  // The observations`
		`  this.allow("createTask";);`
		`  await pendingCreateTask = 0;`
		`  this.allow("dispatchTask";);`
		`  await pendingDispatchTask = 0;`
		`  this.allow("createTask";);`
		`  await pendingCreateTask = 0;`
		`  this.allow("dispatchTask";);`
		`  await pendingDispatchTask = 0`

Fig. 3. Initial configuration and recorded events, and the resulting tester. In this scenario, initial creation of one task results in two observations of (task creation, task dispatch). After each call to `allow`, the tester `await`s until the event is consumed by the model.

5.4 *Implementing the Tester for the Model*

We have seen how the model signals that an event occurs. The tester allows the model to proceed if the same event was observed on the implementation, and waits until the model has actually continued past the event. Most of the tester will consist of a sequence of Creol code like this:

```
this.allow("eventX";);
await pendingEventXCounter = 0;
```

The first line forbids the model to generate an observable event `eventX` that has *not yet* been observed on the implementation. The second line forbids the test case to proceed until the model *has* produced the event `eventX`. Together, these two lines enforce a tight synchronization between the sequence of events as observed on the implementation and on the model.

Of course, if an action by the tester results in an event, the tester has to observe the associated event:

```
t!queue.createTask(taskId);
this.allow("createTask";);
await pendingCreateTask = 0;
```

5.5 *Generating Test Cases*

Testing the implementation against the model consists of:

(i) Designing an initial configuration $conf_I$ (test case input)

(ii) Recording a sequence of observations $events_I$ by running the implementation with the initial configuration

(iii) Translating initial configuration and observation sequence into the model view, resulting in a tester

(iv) Executing the model with the generated tester, reaching a test verdict

Figure 3 shows an example of an initial configuration, the observed events in the implementation ($events_I$) and the corresponding tester.

The initial configuration for the ASK system is created by domain experts, consisting of a task list (stored in the database) and of a set of incoming calls to be simulated by the test driver.

5.6 Reaching a Test Verdict

The instrumented ASK system is started, with the database configuration and telephony environment supplied by the test driver. The result of running the SuT is an event trace $events_I$.

Verifying the implementation's behavior against the model is done with techniques similar to TorX (on-the-fly testing). For technical reasons, it was decided not to run the model and the SuT at the same time, but to replay the traces of the SuT on the model instead.

A test is successful if the model successfully handles the same trace as the implementation and if all assertions and invariants in the model are true. If an assertion in the model is violated, the model itself has an inconsistency and is in error; no verdict about the implementation can be reached. If the tester deadlocks when run in parallel with the model, the implementation violates the test assumption and the test fails. If the tester runs to completion, the test passes.

6 Conclusions and Future Work

Testing multi-threaded implementations is an open field. In our work, we test a multi-threaded implementation against a multi-threaded model. We make use of the fact that Creol's semantics allow for concise modeling, while still being close to a conventional object-oriented imperative programming language. Hence, our model can have a similar structure the implementation. It is our belief that this ease of modeling will encourage developers to use Creol models both during initial modeling and system design, to gain confidence in the system architecture, and as a testing tool to verify the implementation against the model.

We have verified our approach on small test cases and are currently working on developing tool support and integrating the approach into the framework developed within the Credo project [7].

A logical extension of our work would be to influence the implementation's scheduler in order to get more variation in the recorded traces $events_I$. The fault-inducing approach of Edelstein et al. [9] should be straightforward to adopt.

The approach can also be adapted for testing only a part of the system, for example, if the model is incomplete. In that case, some events that originate in a part of the implementation that is not modeled would not be observed in the model and the test would fail. However, if we annotate the origin of recorded events, we can insert actions corresponding to the missing events into the tester; that way, the tester simulates the behavior of the missing parts of the model and the test case can be executed.

Finally, the semantics of Creol allow us very easily to weaken the event execution sequence. This way, we could selectively enable certain reorderings of event observations between model and implementation, e.g. two simultaneous incoming calls could be accepted in different order. It remains to be seen whether this feature results in stronger test cases.

References

[1] Almende website. http://www.almende.com.

[2] ASK community systems website. http://www.ask-cs.com.

[3] ACC: The AspeCt-oriented C compiler. http://www.aspectc.net.

[4] 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 *Testing of Communicating Systems: Method and Applications, IFIP TC6 12th International Workshop on Testing Communicating Systems*, volume 147 of *IFIP Conference Proceedings*, pages 179–196. Kluwer, 1999.

[5] C. Campbell, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes. Testing concurrent object-oriented systems with Spec Explorer. In J. Fitzgerald, I. J. Hayes, and A. Tarlecki, editors, *FM 2005: Formal Methods*, volume 3582 of *LNCS*, pages 542–547. Springer, 2005.

[6] 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, 2002.

[7] Credo: Modeling and analysis of evolutionary structures for distributed services (IST-33826). http://www.cwi.nl/projects/credo/.

[8] 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, 2007.

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

[10] J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho. An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming*, 29(1–2):123–146, 1997.

[11] 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.

[12] J. He and K. J. Turner. Protocol-inspired hardware testing. In *Testing of Communicating Systems: Method and Applications, IFIP TC6 12th International Workshop on Testing Communicating Systems*, volume 147 of *IFIP Conference Proceedings*, pages 131–148. Kluwer, 1999.

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

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

[15] A. Petrenko and N. Yevtushenko. Queued testing of transition systems with inputs and outputs. In R. Hierons and T. Jéron, editors, *Formal Approaches to Testing of Software, FATES 2002 workshop of CONCUR'02*, pages 79–93. INRIA Report, 2002.

[16] R. Schlatte, B. Aichernig, F. de Boer, A. Griesmayer, and E. B. Johnsen. Testing concurrent objects with application-specific schedulers. In J. Fitzgerald and A. Haxthausen, editors, *International Colloquium on Theoretical Aspects of Computing (ICTAC)*, volume 5160 of *LNCS*. Springer, 2008.

[17] 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.

[18] J. Tretmans. Model based testing with labelled transition systems. In R. M. Hierons, J. P. Bowen, and M. Harman, editors, *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*, volume 4949 of *LNCS*, pages 1–38. Springer, 2008.

[19] 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.