Dissertation

Passive Conformance Testing with Concurrent Object-Oriented Software Models

Rudolf Schlatte

Graz, 2010

Institute for Software Technology
Graz University of Technology

Supervisor/First Reviewer: Univ.-Prof. Dipl.-Ing. Dr. techn. Franz Wotawa
Second Supervisor: Dipl.-Ing. Dr.techn. Bernhard K. Aichernig
Second Reviewer: Dr. habil. Martin Steffen

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, _____ _____

Place, Date Signature

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am _____ _____

Ort, Datum Unterschrift

Abstract

Testing, i.e. checking whether a computing system works as expected, has been an integral part of development since the beginning of computer science. Testing is both standard industrial practice and an area of active academic research, with results and experiences being exchanged between theory and practice. The goal of this thesis is to show the possibility and practicality of adapting and reusing an existing behavioral model, written down in a way that is easy to understand by the working programmer, as an aid in the testing phase of a software project. The modeling formalism employed is based on Creol, an object-oriented concurrent executable modeling language, but can be adapted to any language with an operational semantics.

The goal of the thesis was reached with the development of a conformance relation between a model and an implementation that operates with observable events; a method of light-weight instrumentation of both model and implementation; the development of a toolchain for test case generation; and the validation of the approach against an industrial case study. Additional results of the thesis comprise a method of calculating test cases for single components that takes into consideration the internal scheduling within the component; methods for modeling resource constraints (power consumption, bandwidth, memory) for Creol, and an interpreter that implements these resource constraint semantics; and an operational semantics for an extension incorporating models of timed behavior into Creol and the corresponding interpreter.

Zusammenfassung

Software-Testen, d.h. die Überprüfung, ob ein EDV-System wie erwartet funktioniert, ist sowohl ein integraler Bestandteil jedes industriellen Software-Entwicklungsprozesses als auch Gegenstand wissenschaftlicher Forschung. Das Ziel dieser Arbeit ist es, die Möglichkeit und Zweckmäßigkeit der Wiederverwendung von formalen Modellen, die während des Software-Entwurfsprozesses erstellt wurden, während des Testprozesses zu zeigen. Notwendig zur Akzeptanz eines formalen Modells im industriellen Umfeld ist eine ausführbare Semantik und vertraute Syntax der gewählten Modellierungssprache; diese Arbeit verwendet Creol, eine objektorientierte, nebenläufige ausführbare Modellierungssprache, aber die Ergebnisse können mit wenig Aufwand an andere Sprachen mit operationaler Semantik angepaßt werden.

Das Ziel der vorliegenden Arbeit wurde erreicht durch die Entwicklung einer Konformitätsrelation zwischen Modell und Implementierung eines Softwaresystems, die auf Abfolgen von Beobachtungen beruht; die Entwicklung einer minimal invasiven Instrumentierung von Modell und Implementierung zur Aufzeichnung derselben; die Implementierung eines Testfallgenerators zur automatischen Erstellung von Testfällen aus diesen Aufzeichnungen; und die Validierung der Methode mittels einer industriellen Fallstudie. Weitere Ergebnisse der Dissertation sind eine Methode der Berechnung von Testfällen für einzelne Komponenten, die das interne Scheduling von Prozessen innerhalb der Komponente berücksichtigt; Methoden für die Modellierung von Ressourcenknappheit (Stromverbrauch, Bandbreite, Speicher) für Creol und deren Implementierung in einem modifizierten Creol-Interpreter; und eine operationelle Semantik für die Modellierung von Zeit in Creol sowie ihre Implementierung.

Preface

This thesis was written while I was employed by the International Institute for Software Technology of the United Nations University, Macao, to work on the European Union FP7 project "Credo: Modeling and analysis of evolutionary structures for distributed services" (IST-33826) [23]. The goal of the Credo project was to research compositional modeling, testing and validation of software for evolving networks of dynamically reconfigurable components. The main technologies employed were the network interaction language Reo and the object-oriented specification language Creol.

Working with a large team of international researchers of various backgrounds was an immense help to me, as was participating in the cooperation among researchers and the case studies in which the tools and techniques were put to actual use during the project. This thesis uses just a small part of the Credo project: the results obtained for testing software systems against models written in the Creol language. The overall methodology of Credo is presented in [46].

Acknowledgements

Although a thesis serves to showcase a candidate's own research work, it would be an illusion to pretend that scientific work is done in isolation, without collaboration, input or help from colleagues and friends. In this vein, I would like to thank first and foremost the many project partners from the Credo project, without whose work this thesis would not exist: Frank de Boer, whose "hands-off" leadership style lets everyone work to their full potential; Einar Broch Johnsen, who is not only one of the designers of the Creol language, but can also be counted on to provide help and input on whatever problem appears; Andries Stam and Wolfgang Leister, who were the driving forces behind the ASK and BSN case studies, respectively – this thesis has an immense debt on both the models they developed and their experiences in using the language; the co-authors of the publications that comprise the thesis, first and foremost among them my colleague Andreas Griesmayer; and last but not least my advisors Franz Wotawa and Bernhard K. Aichernig, who also led the testing group within the Credo project and was responsible for hiring me to the project, thus providing me a means for writing this thesis.

In Macao, both the scientific and administrative staff were extremely helpful for creating a comfortable and productive working environment. Thanks go to Jeff Sanders, Zhiming Liu, Antonio Cerone and Tomasz Janowski and the members of their groups for many interesting discussions, and to Wendy Hoi,

Coffee Das Dores, Kitty Chan, Alice Pun, Michelle Ho and Sandy Lee for always coping with bravour with whatever administrative demands are set upon them.

Last but not least, I feel a tremendous intellectual debt towards the founding fathers of this field; the weaknesses of this thesis are felt more acutely when reading the works of Tony Hoare, Edsger Dijkstra, Don Knuth and many others. Their intellectual rigor will always provide inspiration and motivation to strive further. Some years ago, I had the good luck to study with Peter Lucas; his lectures and personal example have always stayed with me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Testing, i.e. checking whether a computing system works as expected, has been an integral part of development since the beginning of computer science. Testing is both standard industrial practice and an area of active academic research, with results and experiences being exchanged between theory and practice.

A common theme in testing is the *model*, which is a description (be it explicit or implicitly assumed by the stakeholders) of the "ideal", error-free system that the computing system being tested (also known as "system under test" (SuT) or simply "the implementation") is compared against. This model can be as abstract as "the system does not crash, no matter what the input", as used in robustness testing, or can be a detailed formal model of (some aspects of) the system's desired behavior, as in some branches of functionality testing.

1.1 Motivation of this Thesis

The goal of this thesis is to show the possibility and practicality of adapting and reusing an existing behavioral model, written down in a way that is easy to understand by the working programmer, as an aid in the testing phase of a software project. Many model-based testing approaches rely on models specifically created as test oracles or for test-case generation purposes; the ability of adapting and re-using a model created in the specification / design phase as a test oracle is an obvious advantage.

This goal was reached with the development of a conformance relation between model and implementation relying on observable events, a method of light-weight instrumentation of both model and implementation, the development of a toolchain for test case generation, and the validation of the approach against an industrial case study. Additional results included in this thesis are an approach for single-object or single-component testing against a behavioral specification that includes scheduling behavior, and a method of adding resource constraints to existing functional models in order to validate their functionality in limited evironments, e.g. in embedded systems. The common theme of all these results is showing the reusability and adaptability of existing models for additional purposes – from requirements specification to testing and deployment planning.

A model of a system might have been created during the requirements gath-

ering and design phases of a software development project; the benefits of using formal methods in these phases is well-documented [39]. Nevertheless, under (perceived or real) time pressure, "non-essential" parts of software development, like documentation and testing, tend to get cut back; furthermore, the rigor and attention to detail required by a formal specification can be a frustrating experience sometimes – all the effort of coding without running code. It is my conviction that this effort can be justified by the prospect of re-using the formal model(s) developed in the specification phase as a *test oracle* during and after coding. Hence, a method that allows to adapt, with minimal effort, the behavioral model of a system for testing purposes is desirable in the software development process. This thesis presents such an approach to adapt behavioral models written in the object-oriented Creol language for testing purposes, with the associated theory and tools.

In order to reuse an existing behavioral model for testing, with minimal changes to both the model and the system under test, we use *passive testing*, which does not require controllability of the system. Validating a software system against a model by recording and replaying traces, while not as mainstream as test case extraction from models [53], is nevertheless an active area of research [49, 9, 5, 18, 17]. One contribution of our approach is the method of adaptating a general-purpose executable model as a test oracle, as opposed of the usage of models constructed explicitly for the purpose of validation of run-time behavior.

The approach was validated against a number of case studies in the Credo project, and other test purposes besides functional whole-system testing (e.g. single-object component testing, testing against resource-constrained models) were explored.

1.2 The Credo Project

The Credo project ("Credo: Modeling and analysis of evolutionary structures for distributed services", FP7 IST-33826) dealt with modeling and analysis of distributed software, more specifically

> [...] the development and application of an integrated suite of tools for compositional modeling, testing, and validation of software for evolving networks of dynamically reconfigurable components. [23]

Figure 1.1 gives an overview of the main objectives: the modeling of the behavior of components using *behavioral interfaces*, namely constraint automata [64, 76] and timed automata [37]; the behavior of the connecting network with Reo, a *network coordination language* [6], and an executable model of component behavior with Creol [57]. The project developed methods and tools for checking behavioral equivalence between the different layers.

The lowest layer of Figure 1.1, the concrete implementation of a software system, was not part of the initial project objectives and description of work of the Credo project, but came in as a result of the work done in this thesis. Since one of the case study partners had an actual implementation of the system that was being modeled, the author decided to develop a testing strategy connecting the Creol model and the existing implementation strategy, thus "grounding" the stack of models being developed. This decision was a factor in the positive

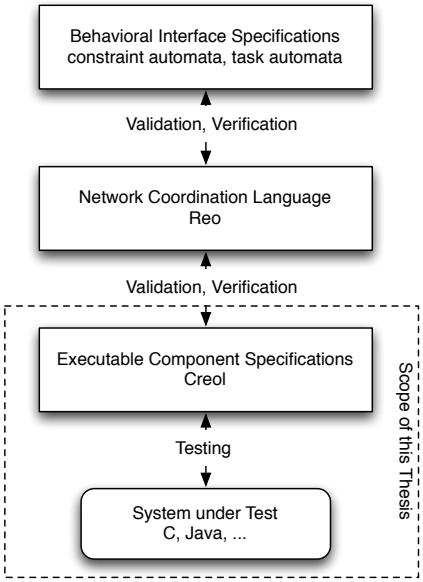

Figure 1.1: Objectives of the Credo project. The work described in this thesis establishes the link between executable component specifications and system under test.

review verdicts of the Credo project – it was generally accepted that Credo was a project with a strong academical / theoretical bias, but the connection from Creol to "real" code, as developed and described in this thesis, was very helpful in addressing comments by the EU-appointed reviewers about applicability of the research results in an industrial setting.

1.3 Main Results

The following results were obtained while working on the publications that form the basis of this thesis:

- The formulation of a theory and technique to test reactive systems against a model using minimally invasive techniques, and without needing to control the environment (model-based grey-box or black-box *passive testing*)
- A tool to generate and execute test cases, implementing the passive testing approach.
- A model checker for the Creol language that is able to handle larger models than using the built-in model checker of the Maude platform.
- An approach for expressing resource constraints for Creol models, a collection of modified Creol interpreters implementing various resource con-

straint semantics, and an approach for testing and quantifying the effects of resource constraints on the model.

As is customary, many of the results were already presented to the scientific community in conferences and workshops before finishing the thesis proper. The following publications contain the main results of this thesis:

- B. Aichernig, A. Griesmayer, R. Schlatte, and A. Stam. Modeling and testing multi-threaded asynchronous systems with Creol. *Electronic Notes in Theoretical Computer Science*, 243:3–14, 2009. Proceedings of the 2nd International Workshop on Harnessing Theories for Tool Support in Software (TTSS 2008).

  The idea and method of adapting an existing Creol model for use as test oracle, as well as the underlying conformance relation, were first developed in this publication.

- B. K. Aichernig, A. Griesmayer, E. B. Johnsen, R. Schlatte, and A. Stam. Conformance testing of distributed concurrent systems with executable designs. In F. S. de Boer, M. M. Bonsangue, and E. Madelain, editors, *FMCO*, volume 5751 of *LNCS*, pages 61–81. Springer, 2008.

  This paper contains a more fleshed-out explanation of the testing approach against executable Creol models, including a description of the tool support implemented since the former paper's publication, as well as a method of generating test inputs from Creol models (developed by A. Griesmayer).

- R. Schlatte, B. K. Aichernig, F. S. de Boer, A. Griesmayer, and E. B. Johnsen. Testing concurrent objects with application-specific schedulers. In J. S. Fitzgerald, A. E. Haxthausen, and H. Yenigün, editors, *ICTAC*, volume 5160 of *Lecture Notes in Computer Science*, pages 319–333. Springer, 2008.

  This paper describes an approach of test case generation that uses single Creol objects as model and includes dealing with scheduler nondeterminism.

- R. Schlatte, B. Aichernig, A. Griesmayer, and M. Kyas. Resource Modeling for Timed Creol Models. *Electronic Notes in Theoretical Computer Science*. Proceedings of the 3rd International Workshop on Harnessing Theories for Tool Support in Software (TTSS 2009) (to appear).

  This paper contains an addition of the main testing approach: how to augment Creol models with models of runtime constraints and test against these augmented models. The paper also motivated work on a timed version of the Creol interpreter, in order to be able to model bandwidth constraints.

The author also cooperated with other scientists at the International Institute for Software Engineering, Macao, with colleagues at Graz University of Technology, and project partners within the Credo project. The following publications, co-written during the time this thesis was under development, are not part of the results presented herein:

- A. Griesmayer, B. K. Aichernig, E. B. Johnsen, and R. Schlatte. Dynamic symbolic execution of distributed concurrent objects. In D. Lee, A. Lopes, and A. Poetzsch-Heffter, editors, *FMOODS/FORTE*, volume 5522 of *Lecture Notes in Computer Science*, pages 225–230. Springer, 2009.

- B. K. 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.

- A. Griesmayer, B. K. Aichernig, E. B. 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.

- T. A. Basuki, A. Cerone, A. Griesmayer, and R. Schlatte. Model-checking user behaviour using interacting components. In *Formal Aspects of Computing*, 21(6):571–588, Dec. 2009.

- I. Grabe, M. M. Jaghoori, B. K. Aichernig, C. Baier, T. Blechmann, F. de Boer, A. Griesmayer, E. B. Johnsen, J. Kleijn, S. Klüppelholz, M. Kyas, W. Leister, R. Schlatte, A. Stam, M. Steffen, S. Tschirner, X. Liang, and W. Yi. Credo methodology. Modeling and analyzing a peer-to-peer system in Credo. *Electronic Notes in Theoretical Computer Science*, 2009. 3rd International Workshop on Harnessing Theories for Tool Support in Software (TTSS 2009). ENTCS, Elsevier, Amsterdam (to appear).

  (While this paper contains the main results of this thesis, its broad scope and the necessarily small contribution each individual part of the Credo project could be afforded did not merit its inclusion in this thesis.)

1.4 Outline of this Thesis

We start by giving a general overview of model-based testing approaches and literature in Chapter 2. Chapter 3 presents the Creol language in detail; understanding the features and theconcurrency model of Creol is necessary for much of the remaining chapters. Chapter 4 presents a method and theory for adapting existing behavioral Creol models as test oracles for passive testing. Chapter 5 expands on the previous chapter, shows how to generate test inputs from a Creol model and how to calculate a model of the environment (based on implementation behavior) to accompany partial models, and presents tools for test case generation and execution. Chapter 6 is a further variation of the approach, this time generating a detailed environment model and scheduling specification in order to test a single component of a Creol model. Finally, Chapter 7 presents results for adding an additional test purpose to the approach: the modeling and testing of resource-constraints (memory, parallelism, bandwidth, battery usage) on the implementation. Finally, Chapter 8 summarizes and concludes the thesis.

Chapter 2

Passive Testing and Model-Based Testing

This thesis explores *passive testing* against *executable models*. This chapter puts the chosen modeling and testing approach in context with the wider field of testing, and presents common terminology and approaches of testing in general and model-based testing specifically.

A software system can be evaluated with respect to a (formal) specification or with respect to the (informal) requirements of its stake-holders. The first activity is called *verification* while the second is *validation*. In the context of this thesis, testing is understood as a verification activity, that is, testing is the process of comparing a system to a (usually higher-level, more abstract) formal description of its desired behavior.

It could be naively assumed that proving programs correct would obviate the need for testing, and indeed, the testing process can only show the presence of errors, never their absence.[1] On the other hand, a proof is only as strong as the assumptions (axioms) it rests upon, such as the assumption that the program will run on faultless hardware. (One of the purposes of stability testing before deployment, "burn-in", is to discover hardware faults in the deployment environment.) Another example for the importance of testing can be found in [40], where the authors show how an algorithm that was proven to be deadlock-free exhibited deadlocks on the deployed system under certain circumstances – the reason being that the proof assumed that creating an arbitrary number of threads was possible. Thus, while the algorithm itself was correct, some of its underlying assumptions were violated by the implementation. In the end, a revised model incorporating this limited number of threads was proved correct under the conditions of the deployment systems, and a working system was implemented. In general, it is prudent to validate the axioms of a proof, and to prove the assumptions of a test [43].

[1] There are, in principle, test suites that show the absence of errors w.r.t. a specification, but they are of infinite size (see [84]).

2.1 Terminology

The software system under assessment is called *System under Test* (SuT). During a test run, the SuT is executed and observed; its behavior generally varies depending on stimuli such as input data and events from the environment. A *test case* is responsible for supplying relevant stimuli to the SuT, observing relevant *test output* behavior, and reaching a *test verdict* (generally, one of Pass, Fail, Inconclusive; the latter resulting from nondeterminism and/or partial models). A *test suite* is a collection of test cases. When a test is executed, the actual output or behavior of the SuT must be compared with the output or behavior expected by the test case. The general problem of determining whether an output is correct with respect to the specification is known as the *oracle problem*; a *test oracle* is an entity that can decide whether a test is successful or not.

Jan Tretmans, in the introduction to [83], defines four areas of testing:

Robustness Testing how does the SuT react if its environment does not behave as expected?

Reliability Testing how long can we rely on the correct functioning of the SuT?

Performance Testing how fast can the SuT perform its tasks?

Conformance Testing does the behavior of the SuT comply with its functional specification?

All these areas of testing need a model to test against. For robustness and stability testing, the model can be quite abstract ("SuT does not crash"), and test cases implement or simulate an environment driving the SuT. Robustness and stability testing differ in that robustness testing needs a more involved model of the environment that considers (an approximation of) all possible inputs, whereas stability testing only needs to consider valid inputs.

Although all testing is testing against a specification or model, the term "Model-Based Testing" (MBT) is usually used in performance and conformance testing, where models are expressive enough to serve for generating test cases, estimating test coverage, and/or checking the validity of the outputs of the system under test, which are the main activities in model-based testing. Hence, research in model-based testing deals mostly with conformance testing and performance testing, since these areas need these more detailed, concrete models of behavior of the SuT.

2.2 Formal Models of Software

The models used in this thesis are written in Creol, an object-oriented, executable modeling language (see Chapter 3). There are many different formal languages to model various aspects of a software system. Hierons et al. [53] give a survey and overview of the major modeling paradigms used for model-based testing. In brief, the following modeling styles are predominantly used in MBT:

Contract-Like Specifications Models of this type have an explicit state, possibly with invariants, and operations that change the state are defined via pre- and postconditions.

Algebraic Specifications In this formalism, a system (classically, but not necessarily, an abstract datatype) is described in terms of the relationships between operations on that type; nothing is said about its internal representation. Gaudel and Le Gall [44] gives an overview of test case generation techniques from algebraic specifications.

Labeled Transition Systems (LTS) Systems are modeled as a (possibly infinite) set of states, with labeled transitions representing input, output or internal state changes. The semantics of many specification languages are based on LTS. Brinksma and Tretmans [13] present an annotated bibliography on testing from LTS, Tretmans [84] a tutorial and introduction that also references some testing tools.

Finite State Machines A system is modeled as a finite set of labeled states, with transitions between states triggered by inputs, and transitions (in the case of Mealy machines) or states (for Moore machines) being annotated by system output. Extensions include hierarchical state (State Charts) and symbolic state, with explicit assignments on transitions (UML state charts). Lee and Yannakakis [60] presents a survey of test case generation methods from FSMs.

Kripke Structures These models are used by model checkers; input is usually in the form of a more readable specification language. Model checkers can be used to automatically generate test cases; for details, see [42, 41].

Dataflow- and Hybrid Models Models of these types are usually used to model embedded and control systems. They are often graphical in nature, executable and have a very natural way of expressing parallel calculations. The general survey of Hierons et al. [53] gives an overview about these kinds of model as well.

As can be expected, there is no one "best" modeling formalism. Education and experience of the modeler, available tools, purpose of the model etc. play a large role in choosing the best modeling language for a given project. The techniques in this work are based on Creol, which is accessible to the working programmer due to its explicit state-based, executable structure.

2.3 Model-Based Testing Approaches

Utting et al. [85] present a taxonomy of model-based testing approaches along various criteria. Of particular interest for our purposes is their criterion of the role that input and output can play in the testing process:

Input-Only Testing In this approach, a formal model is used to calculate *Test Inputs*. The model features a notion of possible inputs (function call, events) for the SuT; input domains are either the same as for the SuT, or an abstraction. Usually some form of *model coverage* is used to determine whether sufficient test input sets have been generated. For example, preconditions can be used to describe allowed input data and to partition the input domain into equivalence classes. Usually then one input is chosen from every equivalence class.

Output-Only (Passive) Testing In this approach, the model is used to validate the system's behavior. This can for example take the form of validating the output of a module against the postcondition of its contract, or trying to reproduce the SuT's behavior in its state machine model. Passive testing is explained in greater detail below.

Input-Output Testing Depending on the model and the testing approach, this can be a straightforward combination of test data generation and output validation using one or multiple models, or take the form of more involved *online testing* approaches, with on-the-fly test data generation and validation. In this case, the model is used to simulate an operating environment for the SuT and controls its behavior.

Passive Testing, also called Runtime Verification, is an interesting area of model-based testing. The basic idea is to *observe* the SuT, record *events* and validate the observed behavior against a model. Passive testing is a complementary method to more common methods of testing; its distinguishing characteristic is that there is no test case supplying inputs to the SuT; rather, the system is observed during normal operation and the formal model serves as test oracle only.

There is a certain similarity in the work of Petrenko and Yevtushenko [69], who separate a test case into queues supplying test inputs on the one hand, and collecting and verifying the SuT's outputs on the other. Passive testing sacrifices all controllability of the SuT, but gains easier applicability especially in an industrial setting, since the active part of the environment (that which supplies stimuli to the SuT) does not have to be modelled or implemented.

A good overview on runtime verification specification languages is contained in [49]. Approaches differ in the formal apparatus employed in the languages (state machines versus regular expressions versus temporal logics), and in the amount of data abstraction assumed. The common theme is that a model is constructed specifically for the purpose of validating runtime traces. One contribution of this thesis is to show a way of adapting an existing executable model, which might be created during specification or system design, to serve as test oracle for passive testing.

Bertolino et al. [11] is similar to this work, with regard to a focus of re-using models for validation purposes at run-time. They generate message sequence charts (MSC) from inter-module communications recorded on the SuT and use them to validate the communication patterns against the system architecture as specified in UML diagrams. This work elides many of their complexities by using event traces directly, without the onus of generating well-formed message sequence charts, and can also be extended to white-box testing (the cited paper talks about problems with their method resulting from programmers communicating with objects directly instead of via the expected interfaces, which in the approach of this thesis is no problem, since the event can simply be recorded within the called object, regardless of how it was called).

A second contribution of this work is a means of generating a model of the environment – the models employed in passive testing have to encompass both the SuT and its environment, in order to validate both inputs to and outputs of the SuT. We show that it is possible to partition the recorded traces into *events*, which should be observed from the model, and *commands*, which should

stimulate the model. In that way, an existing model of the system can be used without the need of a model of the environment.

Chapter 3

The Creol Language and The Case Studies

Creol is a high-level, object-oriented modeling language for distributed and concurrent systems. Creol is formally defined in an operational semantics expressed in rewriting logic [65] that is executable on the Maude [22] rewriting engine. Hence, Creol's definition also serves as an interpreter for models written in the language. Creol is especially suited for modeling loosely-coupled, active and reactive communicating systems and 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.

The language Creol is being developed at the University of Oslo (UiO), and used in education and various research projects, for example the Credo project [23] that funded this thesis. There is a compiler and interpreter [24], an Emacs mode (written by the author of this thesis), and a plug-in for the Eclipse IDE offering editing, compiling and visualization support.

Creol contains the "standard" features of an object-oriented imperative language: inheritance both for interfaces and class implementations; strings and the usual numeric primitive datatypes; tuples; the list, set, and map collection types; assignment, conditional and looping statements. In contrast to, e.g., Java, each Creol object completely encapsulates its state; i.e., all external manipulation of the object state happens through calls to the object's methods. The concurrency model of Creol is based on cooperative scheduling with unspecified message arrival times and scheduling behavior of processes. This allows maximal freedom of execution while still preserving object invariants.

There is also a timed version of the Creol language, with additional language constructs for modeling with discrete time. One of the contributions of this thesis was an operational semantics and Maude interpreter (implemented as enhancement to the existing Creol interpreter) for this timed version of Creol. Timed Creol is described in Chapter 7.

This chapter first gives an overview of the features of Creol that are important to understand the models presented in this work in Section 3.1; for a formal definition of Creol semantics and an accompanying proof system, see [30, 31]. Section 3.2 describes the modeling process and tool support of Creol. Finally, Section 3.3 describes the case studies created within the Credo project, which

are used throughout the thesis.

3.1 Language Definition

3.1.1 Syntax

A Creol model is composed of *classes* that implement *interfaces*; instance variables of classes can be:

- a primitive type (number, string, Boolean),
- a complex type (list, set, map, tuple),
- or a reference to an object.

Interfaces enumerate method names and signatures. Classes implement interfaces and contain methods. Methods can have multiple arguments and return values. Object references are typed with interfaces; class names are used at object creation time only.

The Java-like language syntax of Creol is presented in Figure 3.1. In this overview, we omit some parts of Creol: inheritance and listing the operators of the built-in data types (Float, String, Set, List, Map, etc.), which are standard. For a full overview of Creol, see for example [57].

In the language subset used in the examples of this paper, classes L are of type C with a set of methods $\overline{M}$. Classes can implement zero or more interfaces, which define methods that the class must then implement.

The rest of this section shows the syntax for the elements of the Creol language that are necessary to understand the examples throughout this thesis.

Types

Creol is a strongly, statically typed language – variables contain values of one type, which is known at compile time. Creol has the following types:

- Basic types: strings, integer and floating point numbers, Boolean values
- Lists, sets, maps, tuples
- Labels (future variables) that contain the status and return values of asynchronous method calls
- Interfaces

Expressions

Expressions can be:

- Literals of the supported datatypes (string, integer, floating-point number, Boolean *true* and *false*, tuples, maps and lists).
- The object expressions **`null`**, **`this`** and **`caller`** – the last two evaluate inside method bodies to the object executing the method and the object that sent the method invocation, respectively.

- In timed Creol: the expression **now**, which evaluates to the current value of the global clock.
- Identifiers of local and instance variables, which evaluate to their current value.
- The **new** `C(arguments)` expression, which evaluates to a reference to a fresh object.
- Various composite expressions, for example `a + b` for numerical addition. These expressions are defined in the Creol standard library, in the file `datatypes.creol` of the Creol interpreter.

Statements

The following statements are available:

- **skip**, the no-op statement.
- The suspension statements **release** and **await** *expression*, which suspend the current process and invoke the scheduler. In the case of **await**`(c)` with `c` evaluating to *false*, the process is put into the object's process queue and can be rescheduled only when `c` becomes *true*. If **c** is true, the process is not suspended at all and the await statement becomes effectively a **skip**.
- The assignment statement: *variables* **:=** *expressions* (both single- and multi-assignment are possible)
- The conditional statement: **if** *condition* **then** *statements* **[else** *statements* **] end**
- The loop: **while** *expression* **do** *statements* **end**
- The synchronous method call `obj.method(` *arguments* **;** *results*`)` and the asynchronous method call `l`**!**`obj.method(`*arguments*`)`, where `l` is a label (future variable) for the results.
- The blocking return-value synchronization statement `l`**?**`(`*results*`)`, which receives the results of an asynchronous method call.

These statements are mostly standard fare, apart from the asynchronous method call $l!e.m(\overline{e})$ where l is of type Label and is a reference (future variable) for the return value(s) of m, the blocking read operation $l?(\overline{v})$ that attempts to retrieve the return value from the label, and release points **await** g and **release**.

Guards g on **await** statements 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 is equivalent to **release**, otherwise it is *enabled* and becomes a **skip** (no-op). A **release** statement always suspends the active process and another suspended process may be rescheduled. A suspended process releases its implicit lock on the object's attributes. The *guarded call* **await** $e.m(\overline{e};\overline{v})$ is shorthand for a call which suspends the active process until the reply to the call has arrived; it abbreviates $l!e.m(\overline{e})$; **await** $l?; l?(\overline{v})$.

P::=	$\overline{D}\ \overline{L}$	D::=	**interface** I **extends** $\overline{I}$ $\{\overline{M_s}\}$
T::=	I | **Label**$[\overline{T}]$ | **Bool**	L ::=	**class** $C(\overline{v})$ **begin** $\overline{\textbf{var}\ f:T;}\ \overline{M}$ **end**
	| **Int** | **String** | ...	M::=	**op** m(**in** $\overline{x:T}$ **out** $\overline{x:T}$) == $\overline{\textbf{var}\ x:T;}\ \overline{s}$ **end**
v::=	$f \mid x$	e ::=	v | **new** $C(\overline{v})$ | **null** | **this** | **caller** | ...
g::=	$e \mid l? \mid g \wedge g$	s ::=	$l!e.m(\overline{e}) \mid !e.m(\overline{e}) \mid l?(v) \mid e.m(\overline{e};\overline{v})$ | **await** g
			| $v := e$ | **skip** | **release** | **await** $e.m(\overline{e};\overline{v})$
			| **while** g **do** $\overline{s}$ **end** | **if** e **then** $\overline{s}$ **end**
			| $\overline{s}$ **[]** $\overline{s}$

Elided for brevity: expressions on datatypes (string, numeric, boolean)

Figure 3.1: The language syntax of a subset of Creol. Variables v are fields (f) or local variables (x), C is a class name, I an interface name and l a label (future variable). Some details (e.g. access to attributes shadowed by inheritance) have been omitted.

The `l?()` statement to wait for the return values is *blocking*, which means that the processor is not released and no other 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, which models the "transfer of control" semantics of conventional method calls, can be implemented by calling a method and immediately issuing a blocking wait for its return. (Note that this blocking statement may lead to deadlocks in models.)

Interfaces

Objects are referenced by interface only, never by class; the only time a class is named in code is in a **`new`** statement, but the newly-created instance is assigned to a variable typed by an interface. Interfaces form a semi-lattice with the interface **`Any`** as top element. The syntax for declaring an Interface is:

```
1 interface I [ extends I ]
2 begin
3   with CI
4     method declarations ...
5   ...
6 end
```

Methods declared in the scope of a **`with`** declaration can only be called from objects implementing the *co-interface* CI. The **`caller`** variable will be of type CI.

Classes

The syntax of a class definition is similar to an interface definition, with added elements: methods are defined instead of declared, and private methods and instance variables are defined. A class can have class parameters (constructor arguments), and lists the interfaces that are implemented and the classes that are inherited from, if any.

```
1 class C [ (class_parameters) ]
2   [ implements I ] [ extends C ]
```

```
3 begin
4   instance variable declarations
5   [ op init == statements ]
6   [ op run == statements ]
7   method definitions ...
8   with CI
9     method definitions ...
10  ...
11 end
```

Class parameters are treated as read-only instance variables. Additional instance variables are declared in the class definition body. The **init** method is typically used to initialize instance variables, the **run** method is used to implement active object behavior. These two special methods are run at object creation time. Additional private methods can be defined before the first **with** section.

All method declarations of the interfaces that a class implements must have a corresponding method definition in the class or one of its superclasses, with the same parameter list and co-interface.

Method Declarations and Definitions

Method declarations have the form:

```
1 op method_name (in arguments ; out return_values);
```

Method definitions are of the form:

```
1 op method_name (in arguments; out return_values) ==
2   var local_variable : type [ = expression ]
3   statement_list
```

where the identifiers named in `return_values` are to be used like uninitialized local variables – the values assigned to these identifiers are passed back to the caller when the method finishes.

3.1.2 Special Language Features

Active Objects

Creol objects can be *active* (having a dedicated **run** method that is started upon object creation) or *passive* (only reacting to messages). All method calls (including self-calls) result in a new process to be created within the called object. Each Creol object conceptually contains its own processor and manages its processes by itself, interleaving active and reactive behavior. Processes in different objects execute concurrently, only one process can be active in one object at a time.

Within each object, at most one process is in an *active* or *blocked* state; that process has exclusive access to the object's attributes. The other processes in the object are *suspended*. Some suspended processes are *ready* to run, the others are *waiting* for some condition to become true. In a standard setting, there are no assumptions about the order of process execution within an object. (For an approach to add schedulers to Creol objects, see [72] resp. Chapter 6.)

Inter-Process Communication and Synchronization within an Object

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. This cooperative scheduling semantics has great benefits for modeling: the places where object invariants must be valid are syntactically apparent (namely, release points and method termination), and no explicit locking is necessary to avoid race conditions.

We distinguish between blocking a process and releasing a process. *Blocking* stops the execution of the process, but does not let another suspended process resume. *Releasing* a process suspends the execution of that process and lets another process become active. 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. Processes need not terminate for scheduling to take place – the execution of several processes within an object can be combined using *release points* within method bodies. For example, in Figure 3.2 (right-hand side), the process `m1` dynamically creates a new process `m3`. Since it is an asynchronous call, `m3` does not begin running immediately; rather, `m1` continues after the call until it reaches a release point, at which point the object schedules another thread from its thread pool.

At a release point, which consists of a special statement (**`release`** or **`await`**) in the method body, the active process is *suspended* and *some* suspended process may resume. If the active process tries to read from a future variable whose associated process has not yet finished, the process is *blocked* until a return value can be read. This means no other process within the object is allowed to run in the meantime. Return values can be read from futures in a non-blocking way by suspending the process via an **`await`** statement until the return value is ready.

The local scheduling of processes inside an object is given by Boolean expressions (guards) associated with release points. These guards may depend on local state, allowing cooperative scheduling between the processes within an object, but may also depend on future variables, i.e. the object's communication with other objects. Note that guards are non-monotonic – in general, a guard that depends on object state can become true and then false again. A guard solely depending on future variables is monotonic – once true, it will stay true (in Creol, a method call that has returned can never restart).

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, which is the method of inter-object communication and synchronization.

Communication Between Objects

Communication between objects in Creol is based on method calls. Method calls are a priori asynchronous; in the caller, method replies are assigned to labels (also called *future variables*, see [26]). There is no synchronization associated with *calling* a method, the caller continues execution as normal. However, *reading a reply* from a label is a blocking operation and allows the calling object to synchronize with the callee. A method call directly followed by a blocking read operation models a synchronous call. Thus, the calling process may decide

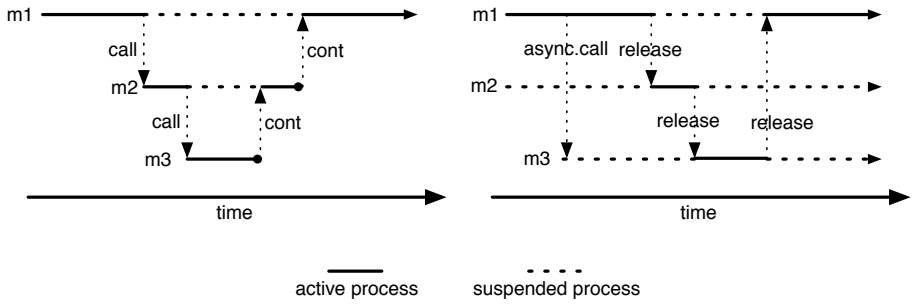

Figure 3.2: Modeling function calls via Creol synchronous method calls (left) and parallelism via Creol asynchronous calls and release points (right).

at runtime whether to call a method synchronously or asynchronously.

The execution model of Creol has been derived from the actor model [52] and uses *cooperative multi-programming* for coordination. For method calls, the caller can choose whether to block and wait for a return value (synchronous call), to synchronize with the callee later (asynchronous call), or to not synchronize at all by ignoring the return value. In the second case a *future variable* [50, 26] is used by the caller to poll the method invocation's termination and to obtain the return values, blocking if necessary. Additionally, the identity and type of the caller is available in most method bodies through the variable **`caller`**, which allows for type-safe call backs. A process need not (and indeed cannot) distinguish whether it was created from a synchronous or asynchronous call. Thus, Creol provides and allows to combine different styles of object interaction.

These features together allow Creol to model both concurrent and single-threaded control flows in a uniform way. Figure 3.2 illustrates the flow of control between 3 processes within one object in the synchronous and asynchronous case. Note that `m3` is called asynchronously in one case and synchronously in the other – a method has no way of determining whether it was called synchronously or asynchronously.

As seen in Section 3.1.1, the syntax for method calls is as follows:

Synchronous call: `object.name(in-parameters;out-parameters)`

Asynchronous call: `future-variable!object.name(in-parameters)`

After a synchronous call returns, the output parameters contain the return value(s) of the method. For an asynchronous call, the return value(s) are stored in the caller's future variable upon completion. If an asynchronous call does not return any value, there is an abbreviated call syntax that is for example used in Line 14 of Figure 3.10.

As a special case, a synchronous self-call unconditionally transfers control to the new process created by the self-call, and arranges for an unconditional transfer of control back to the calling process upon completion (Figure 3.2, left-hand side). Reading from a future variable *blocks* the whole object until the values arrive; hence, a synchronous method call in Creol can be seen as just an asynchronous call plus an immediate blocking read of the associated future variable. For synchronization without whole-object blocking, an **`await`**

statement is used that *suspends* the process but allows other processes in the object to run.

3.2 Modeling with Creol

A Creol model consists of a set of classes. To execute a model, the Creol interpreter is started with a class name (and constructor arguments). The interpreter creates an object of the given class and calls its constructor and **`run`** method. This initial object is then responsible for initializing the model and creating the needed objects. For example, in a model of a sensor network, the initial object will instantiate one object per sensor node that is to be simulated, and connect the sensor objects with each other.

The interpreter terminates after all objects have stopped their work, or at the user's discretion after a defined number of steps. The result is the state of the model, including the internal state of all objects. The model state can be visualized using a tool running on the Eclipse platform.

3.2.1 Levels of Abstraction in Creol Models

Experience gained from using Creol, both in the Credo project and elsewhere, has shown that there is more than one "style" for modeling. Due to the multiple communication patterns possible in Creol, objects tend to fulfil different roles, depending on the abstraction level of the model.

At a low abstraction level, Creol objects are used similar to objects in a conventional Java-style object-oriented programming language: as means of structuring and encapsulating data and the operations working thereon. Models of this type usually resemble in their structure the code of the implementation, both with respect to data structures and control flow. Hence, the predominant communication style is the synchronous method call, which simulates a "conventional" programming thread that executes within different objects as time passes.

More abstract models, on the other hand, use objects more as models of actors in the real world, or of entire components or subsystems of the implementation. The portion of asynchronous method calls is higher, and the data values passed between objects tend to be more abstract.

In summary, it can be said that more abstract Creol models tend to be distributed, with objects modeling physical entities, with more concrete models resembling code in object-oriented programming languages instead. The two case studies of the Credo project illustrate the different styles. While the ASK system model (see Section 3.3.1), especially in its earlier versions, very closely follows the ASK system implementation's architecture and control flow, the BSN sensor model (Section 3.3.2) in the end had one object instance per sensor modelled, plus one object for the air space connecting the sensors. An earlier version of the BSN sensor model that used objects to model the messages passed between sensors was abandoned in favor of using maps as data structures.

Table 3.1 summarizes the two modeling styles. It should be mentioned that these styles should be seen as "emergent properties" of models and not as hard-and-fast rules for modeling – for example, the BSN model, although mostly

Modeling style	low-level	abstract
Method calls	mostly synchronous	asynchronous
Use of objects	for structuring data	for modeling physical entities
Object communication	models control flow	models data flow
Abstraction level	low	high

Table 3.1: Different modeling styles in Creol.

written in the abstract style, contains a very detailed, low-level style implementation of the AODV routing algorithm [68, 67], since that part of the system is of particular interest to the modelers and had to be investigated in greater detail.

3.2.2 Tool Support

During the time of the Credo project, various tools were developed for the Creol language: compiler, interpreter and editing environments.

The Creol Compiler and Interpreter

First and foremost in tool support is a means of executing a language – for Creol, the compiler and interpreter were implemented by Marcel Kyas, with contributions by Ingrid Chieh Yu and Jasmin Christian Blanchette.

The Creol *interpreter* is implemented in Maude and implements an operational semantics of the Creol language as a collection of Maude's term definitions and rewrite rules operating on these terms. A Creol program is, for purposes of the interpreter, a collection of Maude terms describing classes (and the enclosed instance variables and methods). To simulate a running Creol program, a term is inserted that evaluates to an object; the interpreter's rewriting rules take care of executing its `init` and `run` methods, creating other objects and delivering method invocation messages as caused by the initial object.

The Creol *compiler* converts a Creol program written in Creol's concrete syntax (which was described in Section 3.1.1) into the term definitions needed by the Creol interpreter. Also, type-checking is done by the compiler. The interpreter is essentially dynamically-typed – a type error during execution in the interpreter leads to a "hung" state, since none of the rewrite rules apply for an incorrectly-typed term. For example, the Maude equation

```
eq "+" (int(I) :: int(I')) = int(I + I') .
```

applies only if both operands to the **+** operator are actually integers – attempts of adding a string to an integer would lead to an irreducible statement in the interpreter (but will be caught by the compiler while producing the Maude terms before execution time anyway).

Editing Environments

While a text editor is generally sufficient for writing code in any text-based programming language, more specialized editing environments help programmers to be more productive. For Creol, two such environments have been implemented: a plugin for the Eclipse IDE and a major mode for the Emacs text editor.

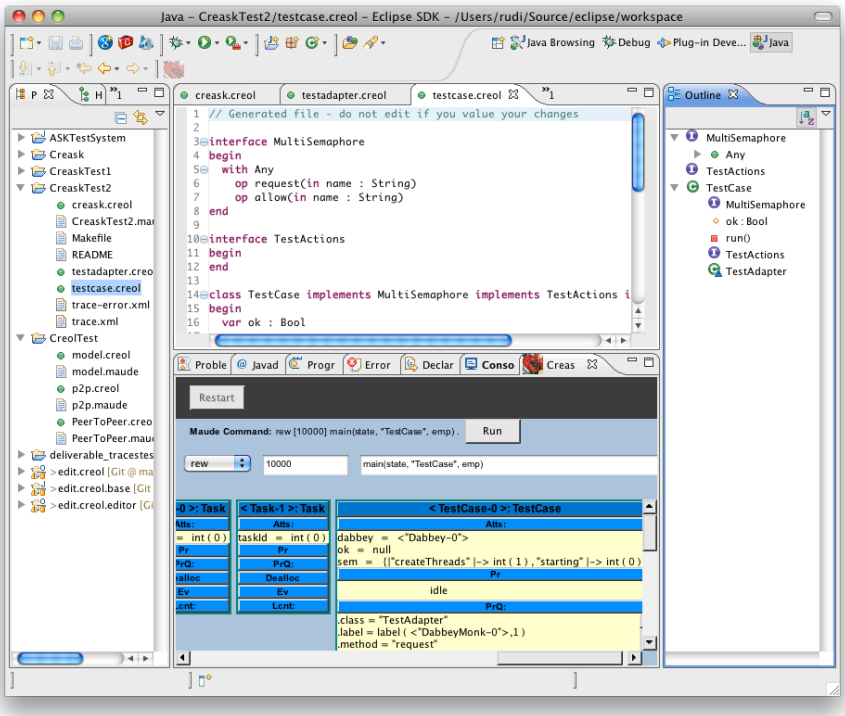

Figure 3.3: Creol support in Eclipse.

The Eclipse plug-in for Creol was developed by Johan Dovland and is currently maintained by the author of this thesis. The Creol plugin supports syntax coloring, folding of class and method bodies, and jumping to class and interface definitions of a Creol file. Also supported are calling the compiler from within Eclipse, and running and stepping through a Creol model with a graphical representation of the model's state. All these features are shown in Figure 3.3.

The Creol mode for Emacs was implemented by the author of this thesis. It supports syntax coloring and indenting of Creol code, calling the Creol compiler and jumping to error locations. Executing a model is supported by running Maude inside an Emacs buffer and interacting with the model there. The Creol mode for Emacs currently does not support a visualization of the model's state while it is running; the modeler has to read the Maude output directly. Figure 3.4 shows a Creol file being edited and compiled within Emacs.

3.3 Case Studies

This section contains a description of the case studies used for obtaining and validating the results of this thesis. Most of the models in this thesis were created in the context of the Credo project. The person responsible for the ASK case study of Section 3.3.1 was Andries Stam of Almende, Wolfgang Leister of NR was in charge of the BSN case study, described in Section 3.3.2.

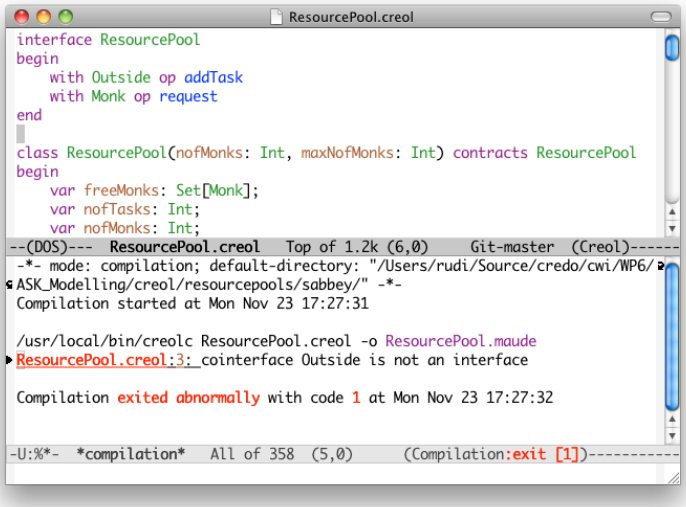

Figure 3.4: Creol support in Emacs.

3.3.1 Case Study 1: The ASK System

ASK is an industrial software system for connecting people to other people via a context-aware response system. ASK was developed by the research company Almende [4] and is marketed by ASK Community Systems [7]. 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 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.

Figure 3.5 shows a simplified architectural view of the existing ASK system. The "heartbeat" of the system is the *request loop*, indicated by thick arrows. A request can be e.g. an incoming phone call from a user of the system or a pre-configured task in the database that is to be executed at a specific time. A request always 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 (Reception, Matcher, Executer, Resource Manager) work together to search for appropriate participants for a request, fill in request details, determine how to connect the participants, or in general figure out the best way in which a request can be fulfilled:

- 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

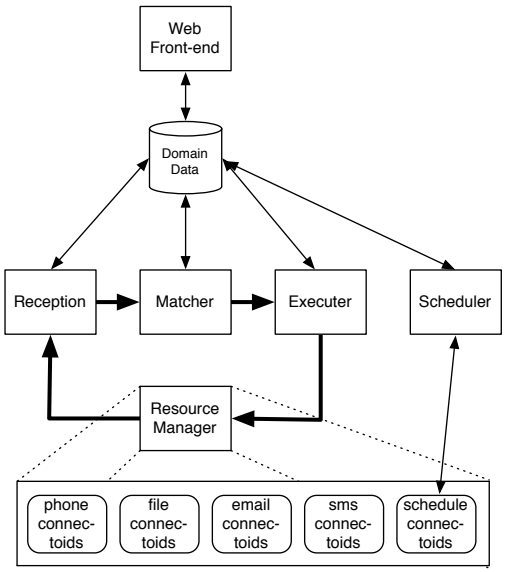

Figure 3.5: Overview of the ASK system architecture.

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 facilitates 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 initial request containing 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, which involves the playing of sound files of the available choices. 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.

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 respectively the number of idle threads.

Two reference models for the ASK system have been developed in Creol, in collaboration with Almende [1]. A low-level model follows the structure of the C implementation closely, while a second, high-level model abstracts away from concrete behavior and is written in a more abstract, nondeterministic style.

An example of a class from the low-level model is given in Figure 3.6, 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 of threads, respectively. The class 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 in Line 9 ff. creates the *balancer* task which is responsible for creating and deleting working threads when needed. The `dispatchTask` method (Line 21) inserts tasks into the task queue that are then executed by an idle worker thread. Method `createThreads`, starting on Line 24, creates a given number of worker threads, which themselves look into the task queue for open tasks. After the system is set up, the thread pool is activated from the outside by the `start` method, which calls `createThreads` with the initial number of worker threads (as set by the class parameter `size`). (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 of the `start` method.)

The same class in the high-level model can be seen in Figure 3.7. While the line count is about the same, that model also contains the functionality of the `TaskQueue` and `Balancer` classes (not shown in the low-level model), and the `Thread` class is about 50% shorter. The main difference is the use of more abstract constructs of the Creol language: for example, while the low-level model replicates the C implementation's search through a linear list of threads, checking whether each thread is busy, the high-level model uses Creol's **`choose`** construct (Line 23) and uses a data structure for free threads only, with threads responsible for inserting themselves into the free set.

3.3.2 Case Study 2: Biomedical Sensor Networks

A wireless sensor network is a wireless network of spatially distributed autonomous devices using sensors to cooperatively monitor physical, environmental or biomedical conditions, such as temperature, sound, vibration, pressure, motion, pollutants or biomedical signals at different locations. Sensor networks have been an active area of research for more than a decade, with applications in e.g. medicine, military, oil and gas, and smart buildings. A biomedical sensor network (BSN) consists of small, low-power and multi-functional sensor nodes that are equipped with biomedical sensors, a processing unit and a wireless communication device. Each sensor node has the abilities of sensing, computing and short-range wireless communication. Due to BSNs' reliability, self-organization, flexibility, and ease of deployment, the applications of BSNs in medical care are growing fast.

The case study presented in this section was developed as part of the Credo project [23]. It models a sensor network consisting of a set of *sensor nodes* and one *sink node*. Sensor nodes are actively monitoring their environment and sending out their measurements. In addition, sensor nodes have the task of routing messages from neighboring sensor nodes towards the sink node. The sink node, typically connected to back-end processing, receives data from all nodes but does not create any data itself.

Connectivity is modeled by a *network object*. This object does not correspond to a physical artifact, but represents the topological arrangement of nodes and their connectivity and also models the behavior of broadcasting a message from a node to its neighboring nodes. Figure 3.8 shows an arrangement of four sensor nodes and one sink node.

In our model, sensor node objects have active behavior: after creation, they transmit a sequence of measurements and then switch to idle (reactive) behavior, only listening for and retransmitting messages.

Figure 3.9 shows the model of a sensor node. Its active behavior is implemented by the **`run`** method starting at Line 16. A sensor node has two functions: read sensor values (method **`sense`**) and send them to neighboring nodes (method **`transmit`**), and receive and re-send values from other nodes in the network (methods **`receive`** and again **`transmit`**). The nondeterministic choice operator (**`[]`**) in Line 19 chooses between reading a sensor value and transmitting a value that can either originate from the node itself or from the network. The method **`receive`** models the receiving part of the node's behavior and is called from the `Network` object.

Figure 3.10 shows the network model. This class does not model a physical object; instead it describes and implements the topology of co-operating `Node` objects; i.e., which other nodes will receive a message broadcast by some node. Line 6 shows the data structure containing the connection map, the method starting in Line 10 implements the network's behavior. The pragma statements in lines 2 and 10 restrict objects of this class to have only one concurrent running `broadcast` method.

```
1  class ThreadPool(size: Int, maxNofThreads: Int)
2    implements 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     var mrate: Int;
11     taskCtr := new Counter;
12     threadCtr := new Counter;
13     busyCtr := new Counter;
14     taskQueue := new TaskQueue(taskCounter);
15     threads := nil;
16     mrate := 5;
17     balancer := new BalancerTask(1, taskCtr, threadCtr, busyCtr,
18       maxNofThreads, mrate, taskQueue, this);
19     this.dispatchTask(balancer;)
20
21   with Any op dispatchTask(in task: Task) ==
22     taskQueue.enqueueTask(task;)
23
24   with Any op createThreads(amount: Int) ==
25     var i: Int;
26     var thread: Thread;
27     i := 0;
28     while (i < amount) do
29       thread := new Worker(taskQueue, busyCtr, threadCtr);
30       threads := threads |- thread; // append thread
31       threadCtr.inc(;);
32       i := i + 1
33     end
34
35   with Any op start ==
36     this.createThreads(size;)
37 end
```

Figure 3.6: ThreadPool of the ASK system, in the low-level model (instantiation of Counter and TaskQueue omitted).

```
1  class ResourcePool(nofThreads: Int, maxNofThreads: Int)
2    implements ResourcePool
3  begin
4   var freeThreads: Set[Thread];
5   var nofTasks: Int;
6   var nofThreads: Int;
7
8   op init ==
9    freeThreads := {};
10   createThreads(nofThreads;);
11   nofTasks := 0
12
13  op createThreads(in nofThreads: Int) ==
14   var thread: Thread;
15   var n: Int := nofThreads;
16   while (n > 0) do
17    thread := new Thread(this);
18    n := n - 1
19   end
20
21  op chooseThread(out thread: Thread) ==
22   await ~ isempty(freeThreads);
23   thread := choose(freeThreads);
24   freeThreads := remove(freeThreads,thread)
25
26  op task ==
27   var thread: Thread;
28   chooseThread(;thread);
29   !thread.task();
30   nofTasks := nofTasks - 1
31
32  op poisonTask ==
33      // omitted for brevity
34
35  op shepherd ==
36      // omitted for brevity
37
38  with Thread op request ==
39   freeThreads := add(freeThreads,caller)
40
41  with Outside op addTask ==
42   nofTasks := nofTasks + 1;
43   !task();
44   !shepherd()
45
46 end
```

Figure 3.7: Thread pool of the ASK system in the high-level model.

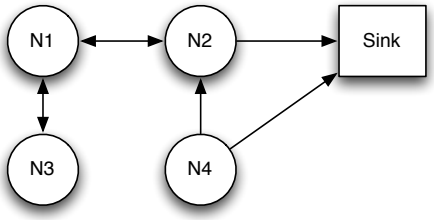

Figure 3.8: A biomedical sensor network: 4 sensor nodes and one sink node. Note that the connections are not necessarily symmetric: Node 4 transmits with more power and can therefore reach Node 2, but Node 2 cannot reach Node 4.

```
1  class SensorNode(id: Int, network: Network)
2  begin
3    var received: List[[Int,Int]] := nil
4    var outgoing: List[[Int,Int]] := nil
5    var noSensings: Int := 3  // No. of sensings to do
6    var seqNo: Int := 0 // Running package seq. no
7
8    op transmit ==
9      !network.broadcast(head(outgoing));
10     outgoing := tail(outgoing)
11   op queue(in data: [Int,Int]) ==
12     outgoing := outgoing |- data
13   op sense ==
14     queue((id,seqNo);); // dummy value
15     seqNo := seqNo + 1
16   op run ==
17      while true do
18        await seqNo < noSensings; sense(;) // read sensor
19        [] // nondeterministic choice
20        await #(outgoing) > 0; transmit(;);
21        release
22      end
23   with Network // receive data from outside
24     op receive(in data: [Int,Int]) ==
25       if ~(data in received) then
26         queue(data;)
27         received := received |- data;
28       end
29 end
```

Figure 3.9: Model of a sensor node. The `receive` method, called by the network, implements reactive behavior, the `run` method implements the node's active behavior.

```
1 class Network
2   pragma Max_resources(1)
3 begin
4   // All nodes in network that have registered and
5   // their connections.
6   var nodesConns: Map[Node, List[Node]] := empty()
7 [...]
8   // Broadcast a message from a node to its neighbors
9   with Node
10    op broadcast(in data: [Int,Int]) pragma Need_resources(1) ==
11      var receivers: List[Node] := get(nodesConns, caller)
12      while ~isempty(receivers) do
13        if head(receivers) /= caller then
14          !head(receivers).receive(data)
15        end
16        receivers := tail(receivers);
17      end
18 end
```

Figure 3.10: Model of the network. (Code to initialize the connection map `nodesConns` elided.) Only one `broadcast` method can be called simultaneously because of the specified resource availability.

Chapter 4

Trace-Based Passive Testing of Creol Models

Modeling concurrent systems and testing multi-threaded systems against models of their behavior is an exciting field of study. This chapter presents an approach for constructing and executing test cases using the Creol language. The approach was validated using a case study consisting of an industrial-size multi-threaded application and a behavioral model written in the Creol modeling language.

Models written in Creol, an object-oriented, concurrent modeling language, tend to be structurally similar to the finished implementation; we show how to exploit this property for re-using existing Creol models as test oracles. Partial models that assume input or stimuli from the environment can be used as well; in this case, a model of the environment is automatically generated from observed runtime behavior of the SuT. We also show the underlying conformance relation between model and system under test.

(The publication which forms the basis for this chapter, "Modeling and Testing Multi-Threaded Asynchronous Systems with Creol" [1], was written with Bernhard K. Aichernig, Andreas Griesmayer, and Andries Stam.)

4.1 Introduction

Formal testing of single-threaded programs can rely on a rich body of theory and industrial experience [43, 86, 35]. Formal testing of multi-threaded or distributed systems, on the other hand, is still an open area of research. This chapter presents work on modeling a concurrent system and testing the system against the model.

As modeling language, Creol [57] 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 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 chapter is to show a way to instrument existing Creol models so they can be used for testing, without needing extensive

restructuring or rewriting. Another contribution of the chapter is to present a conformance relation between a model and an implementation in the face of minimal controllability of both implementation and model.

We validate our testing approach through the 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. See Chapter 3, Section 3.3.1 for more information about the case study.

The testing approach in this chapter relies only on the SuT's observable behavior. No test input needs to be generated; instead, relevant parts of the recorded behavior of the SuT are used to generate a model of the environment to use during validation of the SuT's behavior.

The rest of this chapter is organized as follows: Section 4.2 presents some related work in the area of formal testing. Section 4.3 contains a short discussion of the ASK case study in the context of this chapter, Section 4.4 presents the conformance relation used to generate test cases. Section 4.5 describes the approach for generating test cases, adding instrumentation to model and implementation, and reaching test verdicts. Finally, Section 4.6 contains conclusions and discusses possible future work.

4.2 Related Work

There is considerable previous work on the use of formal methods for testing components [43, 83]. 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 [35], TestGen [51] and TorX [10].

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 [86]) 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 [69] 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 [84]). 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 [14], 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.

4.3 Case Study Scenario

The scenario that we use as an example throughout this chapter models the creation, dispatch and execution of tasks inside the ASK components. However, the introduction to the ASK system in Chapter 3 already shows that more complex scenarios are possible by modeling and instrumenting other parts of the system.

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.

4.4 A Conformance Relation for Passive Testing

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 practical. 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. On the other hand, a complete pre-computed test graph would be enormous due to the possible interleaving inherent to concurrent systems. Instead, we let the system run by itself and use the Creol model as oracle for the test run. To test if an execution of the SuT is valid, the tester tries to reproduce it in the model too – only if that is possible, the run is valid.

In the following we assume that we have a Creol model that completely models the SuT and has a similar structure.[1] Due to their similar structure, both model and SuT can be annotated with the same events (modulo an abstraction function π) (see Subsection 4.5.2 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 and, in the case of M, Creol's other nondeterministic statements.

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

$$\begin{array}{ccc} conf_M & \xrightarrow{M} & \{events_M\} \\ \uparrow \pi & & \uparrow \pi \\ conf_I & \xrightarrow{I} & \{events_I\} \end{array} \tag{4.1}$$

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:

$$\forall conf_I \cdot \pi(I(conf_I)) \subseteq M(\pi(conf_I)) \tag{4.2}$$

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 T, a process actor who supplies the initial configuration to the model and restricts the order of observable events during execution of the model. Formally, T is constructed from a specific event trace, and is just the process that emits $\pi(events_I)$ in sequence:

$$events_I \rightsquigarrow T \tag{4.3}$$

[1] Where by "similar structure" we mean that roughly the same traces can be emitted by model and SuT; in particular, the same sets of events must be observable.

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. Formally, this can be seen as the parallel composition of model and tester: $T||M$. The test process can be described as follows:

$$\forall events_I \in I(conf_I) \cdot \pi(events_I) = (T||M)(\pi(conf_I)) \tag{4.4}$$

Section 4.5 describes the implementation of this test approach.

4.5 Test Implementation

4.5.1 Actions and Events: Generating a Test Environment

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** [83] and indeed much of the formal testing theory.

In our case, the situation is slightly different. Like in [69], 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 SuT without being the direct consequence of an action by the tester as well – for example, in the ASK system one task can start another; this means a task scheduling event will be observed without a preceding action by the environment. 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.

4.5.2 Adding Instrumentation to Model and SuT

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.

Instrumenting the Implementation

There are various methods for adding instrumentation to the implementation, depending on circumstances. Groce et al. [49] recommend re-using existing

logging output, which, in their application, is already implemented and budgeted (memory- and processor-wise). In general, if the system already contains logging and diagnostic output, it is easiest to re-use this for testing purposes.

If no or only ad-hoc logging is implemented, adding tracing behavior via other means can be less work. There are some low-level tracing tools available. In principle, the debugging interface of the operating system (for example, the `ptrace` system call on Linux and other Unix-like platforms) can be used to inspect the system's memory and set breakpoints, but this is a quite low-level approach to logging. For example, the Linux manual page for dtrace warns:

> This page documents the way the ptrace() call works currently in Linux. Its behavior differs noticeably on other flavors of Unix. In any case, use of ptrace() is highly OS- and architecture-specific.
>
> The SunOS man page describes ptrace() as "unique and arcane", which it is.

Slightly higher-level logging tools include SystemTap [79] on Linux and dtrace [78] on the Solaris and Mac OS X platforms. These tools work with trace points that are inserted into the source by the developer at well-chosen points, but they can also intercept arbitrary function calls. Logging or tracing is specified via a script; i.e. what happens at a specific trace point is not written down in the source code itself.

Another possibility for tracing is using aspect-oriented programming, which works by augmenting certain points in the program ("point cuts") with behavior that is orthogonal to the main purpose of the code at that point, for example logging or locking functionality. We used Aspect-C [8] 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, or by the system's users should the real system's behavior be recorded. Typically, the following events are logged:

- 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. In our case, we recorded creation and termination of worker threads and beginning and end of tasks.

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 4.4 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.

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

Figure 4.1: 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.

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:

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

Figure 4.1 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.

4.5.3 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 pairs of Creol statements like these:

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

These two lines synchronize with an **await** `tc.request("eventX")` line in the model. The first line allows the model to generate an observable event `eventX` that has been observed on the implementation. The second line forbids the test case to proceed until the model has produced that expected event. Together, these two lines enforce a tight synchronization between the sequence of events as observed on the implementation and on the model.

4.5.4 Generating Test Cases

Testing the implementation against the model consists of:

1. Designing an initial configuration $conf_I$ (test case input)

2. Recording a sequence of observations $events_I$ by running the implementation with the initial configuration

3. Translating initial configuration and observation sequence into the model view, resulting in a tester

4. Executing the model with the generated tester, reaching a test verdict

Configuration	Events	Tester

createTask	createTask
	↓
	dispatchTask
	↓
	createTask
	↓
	dispatchTask
		`op run ==`
		`  // The initial configuration`
		`  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;`
		`  ok := true`

Figure 4.2: 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 `awaits` until the event is consumed by the model.

Observation	Verdict	Diagnosis
Tester finishes	Pass	
Tester deadlocks	Fail	Model and SuT differ in behavior
Model assertion violated	Fail	Internal model error

Table 4.1: Test Outcomes.

Figure 4.2 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.

4.5.5 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 $events_I$.

A test is successful if the model successfully handles the same trace as the implementation and if all assertions and invariants in the model hold during the test run. 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. Table 4.1 summarizes the possible outcomes and the accompanying test verdicts.

Since Creol's scheduling and some other language features are nondeterministic, observing a deadlock does not necessarily mean the test failed. For example, if two methods are invoked at the same time, the interpreter chooses an arbitrary sequence of process creation, which leads to an arbitrary order of

events occuring in the model. This might lead to spurious test failures, where a different ordering of method invocations would lead to test success.

The `ok` variable in the tester was introduced to have an easy target for model checking $T||M$. Since the model checker of the underlying Maude rewriting engine does not know about Creol semantics and checks possible interleavings of rewrite rules that cannot change the eventual outcome, a specialized model checker was written that only explores Creol's nondeterministic statements. It is implemented as a depth-first search that assumes a terminating model (no infinite runs), which is guaranteed by the test case, which either terminates or deadlocks before terminating, so if no part of the model runs without synchronizing with the test case, the model will terminate. The depth-first search results in memory usage which is not higher than during a normal program run.

4.6 Conclusions

Testing multi-threaded implementations is still an open field of research. In our work, we test a multi-threaded implementation against a multi-threaded model. We make use of the fact that Creol's semantics allows for concise modeling, while still being close to a conventional object-oriented imperative programming language. Hence, our model can have a similar structure as 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.

A possible approach for recording more varied event traces $events_I$, and hence obtaining more test cases, is described in Edelstein et al. [32]. The central observation in that work is that, contrary to conventional wisdom, operating system schedulers are *largely deterministic*. This observation explained hard-to-reproduce errors that only occur on production but not in the test environment: heavy load induces changes in scheduler behavior, thus exposing race conditions – lighter system load, as on a developer machine, makes the error unobservable again. The paper's solution for reproducing these kinds of bugs is to artificially introduce more nondeterminism in scheduling, by instrumenting either the program or, for Java programs, the virtual machine. Making the scheduler truly nondeterministic in that way reportedly helps uncover these hard-to-reproduce bugs, and should be straightforward to adopt for our purpose of obtaining more varied event traces as well.

Our 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. Chapter 5 describes how to create a tester environment based on a recorded event trace in detail.

Finally, the semantics of Creol allows 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 in the model without leading

to test failure. It remains to be seen whether this feature results in stronger test cases.

Chapter 5

Test Input and Tester Environment Generation

This chapter 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.

(The publication which forms the basis of this chapter, "Conformance Testing of Distributed Concurrent Systems with Executable Designs" [2], was written with Bernhard K. Aichernig, Andreas Griesmayer, Einar Broch Johnsen and Andries Stam. The work on dynamic symbolic execution is mainly due to Andreas Griesmayer.)

5.1 Introduction

The method described in this chapter 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.

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 chapter are based on the object-oriented modeling language Creol, a language designed to model concurrent and distributed 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 [65]. 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 (see Section 3.3.1), 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 results of this chapter are:

- A tool-supported method for calculating optimal-coverage test cases from a model that serves as a test oracle.
- An extension of dynamic symbolic execution 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 chapter is organized in two main parts: Section 5.2 gives an in-depth overview of the approach to test input generation using dynamic symbolic execution, Section 5.3 shows how to generate full test cases and calculate test verdicts by recording an implementation's behavior responding to these test inputs, and checking whether the Creol model can exhibit the same behavior. Sections 5.4 and 5.5 contain related work and a conclusion to the chapter.

5.2 Finding Test Cases with Dynamic Symbolic Execution

As explained in Chapter 3, 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.

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 5.2.3. 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* $_S$ *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*

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

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 is determined by the desired kind of coverage (branch coverage, path coverage, statement coverage), 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 [54]) or SMT-solvers (e.g., Yices [90], Z3 [28]).

5.2.1 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 execution 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 $\overline{v} := \overline{e}$ is straightforward: Because fields and local variables in different processes can have the same name and statements from different objects are interleaved, the variable names are expanded to 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): $o_1 \overset{l}{\rightharpoonup} o_2.m(\overline{e})$ represents the *call* of method m in object o_2 from object o_1 with arguments $\overline{e}$; $o_1 \overset{l}{\rightarrow} o_2.m(\overline{v})$ represents the moment when a called object starts execution, where $\overline{v}$ are the local names of the parameters for m; $o_1 \overset{l}{\leftarrow} o_2.m(\overline{e})$ represents the emission of the return values from the method execution; and $o_1 \overset{l}{\leftharpoonup} o_2.m(\overline{v})$ represents the corresponding reception of the values. These four events fully describe method calling in Creol. In this execution model the events reflecting a specific method call always appear in the same order, but they can be interleaved with other statements.

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

Conditional statements in Creol are side effect free, 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.

5.2.2 Test Case Generation

The low-level model of the ASK system (see Section 3.3.1) forms the basis for this chapter's work on testing. Note that in this chapter 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.

To generate test cases from the Creol model, we extend dynamic symbolic execution from Section 5.2 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 $t_{\mathcal{S}}$ 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 $t_{\mathcal{S}}$ that can repeat the same execution path. Only one test case that fulfills $\mathcal{C}$ is required. A new test case is then chosen 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 condition 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.

5.2.3 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 [57], defined in rewriting logic [65] and implemented in

$$\overline{v} := \overline{e}; \overline{s}\big[\Theta, \sigma, \mathcal{C}\big] \Longrightarrow \overline{s}\big[\Theta, \sigma \uplus \langle \overline{v} \rhd (\overline{e}\sigma)\rangle, \mathcal{C}\big]. \quad \text{(ASSIGN)}$$

$$o_1 \xrightarrow{l} o_2.m(\overline{e}); \overline{s}\big[\Theta, \sigma, \mathcal{C}\big] \Longrightarrow \overline{s}\big[\Theta \uplus \langle l \rhd \overline{e}\sigma\rangle, \sigma, \mathcal{C}\big]. \quad \text{(CALL)}$$

$$o_1 \xrightarrow{l} o_2.m(\overline{v}); \overline{s}\big[\Theta, \sigma, \mathcal{C}\big] \Longrightarrow \overline{s}\big[\Theta, \sigma \uplus \langle \overline{v} \rhd l\Theta\rangle, \mathcal{C}\big]. \quad \text{(BIND)}$$

$$\langle g\rangle; \overline{s}\big[\Theta, \sigma, \mathcal{C}\big] \Longrightarrow \overline{s}\big[\Theta, \sigma, \mathcal{C}^\frown\langle g\sigma\rangle\big]. \quad \text{(COND)}$$

Figure 5.1: Rewrite rules for symbolic execution of Creol statements.

Maude [22]. 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 [65]. The rules are presented here in a slightly simplified manner to improve readability.

Denote by $\overline{s}$ the representation of a sequence of program statements. Let $\sigma = \langle v_1 \rhd e_1, v_2 \rhd e_2, \ldots, v_n \rhd e_n\rangle = \langle \overline{v} \rhd \overline{e}\rangle$ be a map which records *key–value* entries $v \rhd e$, where a variable v is bound to a symbolic value e. The value assigned to 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 $\overline{e}\sigma$ denote the application of the parallel substitution to every expression in the list $\overline{e}$. 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 $\mathcal{C}^\frown c$.

The *configurations* of the rewrite system for dynamic symbolic execution are given by $\overline{s}\big[\Theta, \sigma, \mathcal{C}\big]$, where $\overline{s}$ 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 $\overline{s}$ (as described in Section 5.2.1) is in fact generated on the fly by the concrete rewrite system for Creol executed in parallel with the dynamic symbolic execution. Thus, the *rules* of the rewrite system have the form

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

The primed terms on the right-hand side are updated results from the execution of the rule. The rules are given in Figure 5.1 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 $\neg g$.

5.2.4 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 3.6 to compute the number of worker threads to create or destroy depending on a given maximal number of threads, the currently existing number of threads and the number of remaining tasks. Figure 5.2 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 $maxthreads_{\mathcal{S}}$, 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 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 Line 14.

Figure 5.3 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

```
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          && ((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
```

Figure 5.2: 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.

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 $_{\mathcal{S}}$. This enables a flexible monitoring of symbolic values of variables at any arbitrary level in the code.

The test case setup of Figure 5.3 uses two symbolic variables as parameters: the maximum number of working threads $maxWorkThreads_{\mathcal{S}}$ and the initial number of threads $nthreads_{\mathcal{S}}$. 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 $maxWorkThreads_{\mathcal{S}}==0$ and $nthreads_{\mathcal{S}}==1$. 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 Line 28 of Figure 3.6 and correspond to one loop traversal in which a thread is created. The third condition corresponds to Line 7 in Figure 5.2 and shows that the path was taken because $0 >= maxWorkThreads_{\mathcal{S}}$ 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 $maxWorkThreads_{\mathcal{S}}$. For easier presentation, we will exploit this fact in the following and compute new values only for this input and leave $nthreads_{\mathcal{S}}$ constant. Note that this is generally not the case, conditions that rely on several 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 $maxWorkThreads_{\mathcal{S}}==15$, which records the conditions:

```
1 class Main(nthreads_S: Int, maxWorkThreads_S: Int)
2 begin
3   var threadpool: ThreadPool;
4   var executionCounter: Counter;
5
6   op init ==
7     threadpool := new ThreadPool(nthreads_S, maxWorkThreads_S);
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
```

Figure 5.3: Setting up a model for DSE. Here, $nthreads_S$ is the number of initial threads to be created and $maxthreads_S$ is the maximal size of the thread pool.

```
{"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 5.2. The number 10 in the second condition reflects that we create ten tasks at initialization in Figure 5.3. The path condition reflects that all inputs with $maxWorkThreads_S$ >= 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 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 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 Line 28 of Figure 3.6. 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

$maxWorkThreads_{\mathcal{S}}$	$nThreads_{\mathcal{S}}$	Condition
0	1	$nThreads_{\mathcal{S}} > 0 \wedge \neg(nThreads_{\mathcal{S}} > 1)$ $\wedge\neg(maxWorkThreads_{\mathcal{S}} > 0)$
15	1	$maxWorkThreads_{\mathcal{S}} > 0$ $\wedge\neg(maxWorkThreads_{\mathcal{S}} < 10)$
5	1	$maxWorkThreads_{\mathcal{S}} > 0$ $\wedge maxWorkThreads_{\mathcal{S}} < 10$
5	0	$\neg nThreads_{\mathcal{S}} > 0$

Table 5.1: The calculated test input parameters. Every value for $nThreads_{\mathcal{S}}$ besides zero and one leads to another, different path condition, with the resulting test cases differing in the number of initial threads only.

be taken if $(0 < nthreads_{\mathcal{S}})$ and $(1 \geq nthreads_{\mathcal{S}})$, i.e., $nthreads_{\mathcal{S}} == 1$. Thus, using this condition for test case selection, we need a test case for each value of $nthreads_{\mathcal{S}}$, 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, $(0 < nthreads_{\mathcal{S}})$. Thus, we get a new test case with $nthreads_{\mathcal{S}}$==0 (we keep the value $maxWorkThreads_{\mathcal{S}}$==5 from the previous test case). The path condition only consists of:

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

None of the conditions of Figure 5.2 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 $nthreads_{\mathcal{S}}$>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 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.

Table 5.1 summarizes the calculated test input values and their path conditions. The computation of the values for $maxWorkThreads_{\mathcal{S}}$ can be automatized by constraint- or SMT solvers. For the example above we used Yices [27], which takes the negated path condition as input and computes a valuation for the variables if it is satisfiable.

5.3 Test Case Execution

The previous section explained how to calculate test inputs for the implementation that cover different parts of the model. 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 explained in the following section, our test assumption is that a sequence of events that is observed on the implementation can be reproduced (replayed) by the model.

5.3.1 Conformance Testing Using Recorded Event Traces

In the setting of asynchronous, concurrent systems, and when facing nondeterminism, 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 5.1:

$$\begin{array}{ccc} conf_M & \xrightarrow{M} & \{events_M\} \\ \downarrow \rho & & \uparrow \alpha \\ conf_I & \xrightarrow{I} & \{events_I\} \end{array} \tag{5.1}$$

In the literature this is also called U-simulation [29]. 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 5.2 shows the formulation of this trace inclusion relation:

$$\alpha(I(\rho(conf_M))) \subseteq M(conf_M) \tag{5.2}$$

Section 5.3.3 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.

5.3.2 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 [8] 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 5.4 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

```
<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>
```

Figure 5.4: 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.

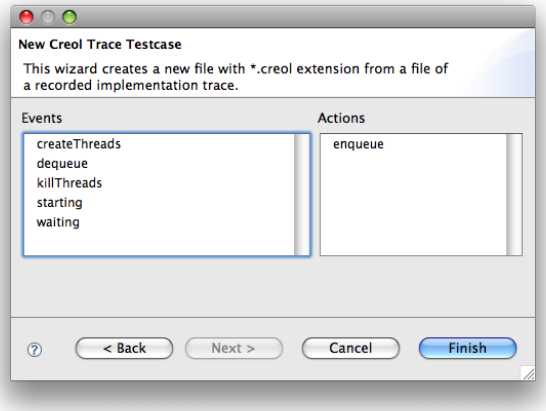

Figure 5.5: Generating the tester from a recorded trace: separating Actions and Events. "enqueue" is to be triggered by the tester, so is designated to be an Action using this dialog.

when a new Task is created) and `dequeue` (recorded when a thread starts working on a task).

5.3.3 Generating the Test Driver and Adapting the Model

As mentioned in Section 5.3.1, 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 5.5 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

```
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
```

Figure 5.6: Test actions interface and test adapter class template, created from the implementation trace. The method bodies must be implemented by the test engineer.

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 5.6 shows the interface `TestActions` and class `TestAdapter` that are generated using the choices made in Figure 5.5. 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 5.6, Line 9) and any actions (line11).

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 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 5.2:

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

Figure 5.7 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 5.4. 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

```
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
```

Figure 5.7: Replaying the trace of Figure 5.4: tester event and action behavior in the model.

to occur, the second statement (`await get(...)`) blocks the tester until the event actually occurs in the model.

5.3.4 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 5.7), 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 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 still inconclusive, since a different scheduling in the model (or executing a different branch of a nondeterministic choice statement) would potentially allow 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".

The first approach was to simply use Maude's built-in model checker – this, however, did not prove feasible for even moderately-sized models. The reason is that the highly-parallel design of Creol (concurrent processes inside objects) leads to lots of possible interleaving of statement execution that *cannot make a difference in the result* but still will be checked by a naive model checker. In practice, the ASK case study + test case could be model-checked within reasonable time to a depth of 500 rewrite steps, which is not sufficient to get to

the end of a test case.

However, in [3] a technique is shown how to reduce the model checking complexity. The basic idea is that most interleavings of statement execution cannot lead to a different end state, except for:

- Scheduling: a model checker must explore executing each enabled process at each scheduling decision
- The non-deterministic statement `[]`: a model checker must explore both alternatives, should they be enabled
- The `choose` function, which non-deterministically chooses an arbitrary element from a set: in this case, each choice must be considered.

We have implemented a simplified model checker that only considers these three decision branches, and could execute test cases to reach a definitive *Fail* verdict.

5.4 Related Work

To our knowledge, the first to use symbolic execution on single runs were Boyer et al. in 1975 [12] 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. [45], and the CUTE and jCUTE tools from Sen at al. [75]. Perhaps the most prominent and most widely used tool in that area is PEX by Tillmann et al. [81], 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 [48, 47], recent work on examining all relevant interleavings in [3].

The use of formal models for testing has a long history, some of the more influential work are [43] and [83]. Various conformance relations have been proposed. They place varying demands w.r.t. controllability and observability placed on the SuT; for example *ioco* [86] by Tretmans et al. demands that implementations 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 [10] and TGV [34], work by simulating a user of the system, controlling input and checking output. A testing method similar to the one described in this chapter, also relying on event traces, was developed by Bertolino et al. [11], 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.

5.5 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.

Chapter 6

Single-Object Testing with Application-Specific Schedulers

In this chapter, 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.

(The publication which forms the basis of this chapter, "Testing Concurrent Objects with Application-Specific Schedulers" [72], was written with Bernhard K. Aichernig, Frank de Boer, Andreas Griesmayer and Einar Broch Johnsen. The formalization of weakest precondition calculation for test input generation was done by Andreas Griesmayer.)

6.1 Introduction

In this chapter 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 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

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 chapter 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 [57, 26], 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 [65] and executes on the Maude platform [22]. 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 chapter 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 [56]. 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.

Chapter overview. The rest of this chapter is organized as follows: Section 6.2 gives a high-level overview and scope for our approach to testing. Section 6.3 explains the modeling approach used, including the high-level specification of scheduling policies. Section 6.4 discusses the details of test case generation and execution. Finally, Section 6.5 discusses related work and Section 6.6 concludes the chapter.

6.2 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. The test input is 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.[1]

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.

6.2.1 Example.

We consider a version of barrier synchronization given by the class `Batch_queue` in Figure 6.1. In a `Batch_queue` object, clients are processed in batches (of size `batch_size`, 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

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

```
1 Interface Client
2 begin
3 end
4
5 class Batch_queue(batch_size: Nat)
6 begin
7  var wc: Nat // number of waiting clients
8  var comein: Nat // number of clients to be processed
9  var display: List[Client] // queue of registered clients
10
11  op init ==
12   wc := 0;
13   comein := 0
14
15  with Client
16   op register ==
17    wc := wc + 1;
18    if wc >= batch_size then
19     comein := batch_size
20    end;
21    await comein > 0;
22    comein := comein - 1;
23    wc := wc - 1;
24    display := display |- caller
25 end
```

Figure 6.1: Motivating example: The `Batch_queue` class.

operator on sequences). Before any call to `register` will return, the object will contain `batch_size` processes.When enough calls are waiting to be registered, the next batch of processes may proceed by assigning the value of `batch_size` 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.

6.3 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 Rusu et al. [71], by 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:

- 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

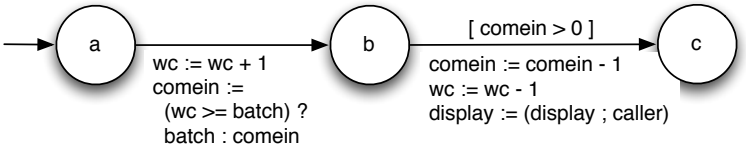

Figure 6.2: Method Automaton of the `register()` method.

- 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[2]. 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 $eval(g, v) = true$.

6.3.1 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 $\langle m, Q_m, q_0^m, T_m, V_m, Val_m \rangle$ 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 is the same as that of 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 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. 6.2 are not prefixed in this way, since every method automaton has access to the same instance variables.

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

6.3.2 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*

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

The semantics of executing a transition of the system automaton is that of executing the transition of *one* of the participating method automata ($q^{m_i} \rightsquigarrow q'^{m_i}$), 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.

6.3.3 Modeling Schedulers: The Scheduler Automata

The system automaton as defined in Section 6.3.2 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, restricts 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*

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

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 $ready(m_i) \in G(A)$ and $\neg ready(m_i) \in G(A)$ 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 $ready(m_i)$ denotes a predicate which is *true* whenever the method automaton m_i has at least one

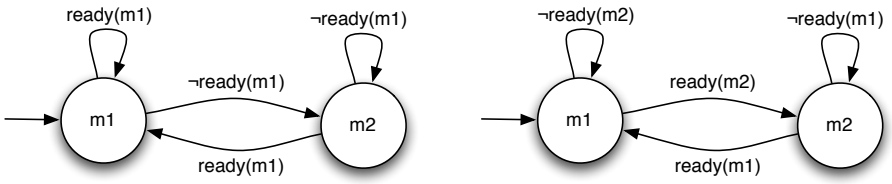

Figure 6.3: Example scheduler automata: priority (left), round-robin (right).

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 6.3 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.

6.3.4 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 $q = \langle m_k, \dots \rangle$ 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 $q_1 = \langle m_i, \dots \rangle$. 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 subexpressions $ready(m_x)$ 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 $A = \langle Q, q_0, T, V, Val \rangle$ be a system automaton for the methods $m_1, \dots, m_n$. For a state $q \in Q$ and a scheduler guard $g \in G(A)$, scheduler guard expansion is a function $[\![g]\!]_q$, inductively defined as follows:*

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

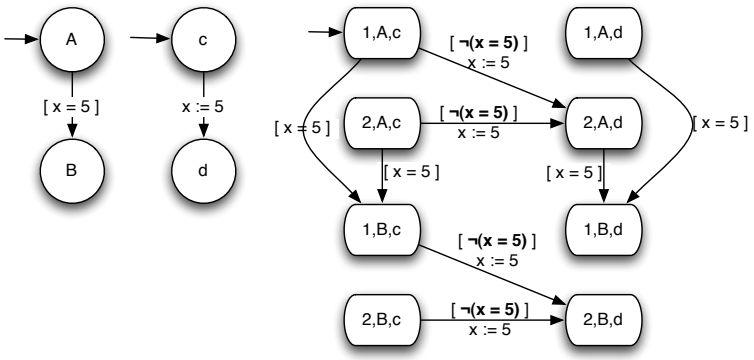

Figure 6.4: Two simple method automata and a system automaton consisting of the two automata running in parallel under the priority scheduler of Figure 6.3 (guards in bold added by the scheduler).

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*

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

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.

6.4 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 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 [55]:

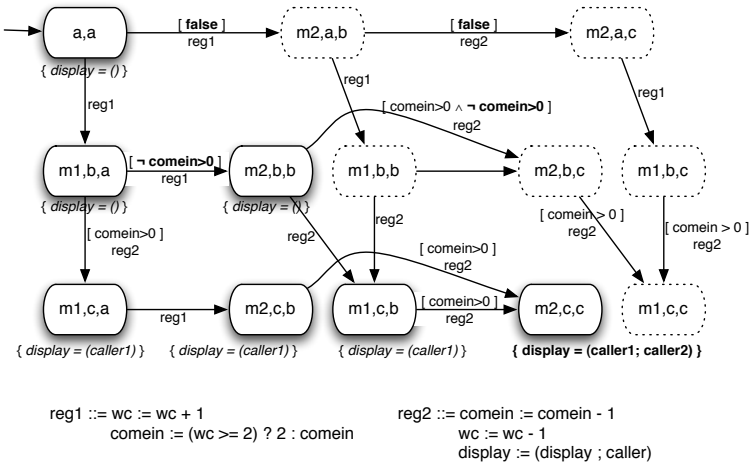

Figure 6.5: A scheduled system automaton with two method automata for the `register` method, under priority scheduling and with batch size 2. Guard terms in bold are added by the scheduler, states that are unreachable under priority scheduling are dashed.

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 6.5 shows the graph of a system automaton that models two invocations of the `register` method and `batch_size` size `2`, scheduled with the priority scheduler from Figure 6.3. 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 6.5 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 *display* = (*caller1*; *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[3]. The actual implementation must not block for this input and must satisfy the assertion.

[3]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.

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:

$$c_p(q) = \bigvee_{\forall\langle q,g,S,q'\rangle\in T} wp\,(S, c(q')) \wedge g \tag{6.1}$$

$$c_{\neg p}(q) = \bigvee_{\forall\langle q,g,S,q'\rangle\in T} wp\,(S, \neg c(q')) \wedge g \tag{6.2}$$

$$c(q) = c_p(q) \wedge \neg c_{\neg p}(q) \tag{6.3}$$

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 **AF** p [21]), 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 6.5 consists of the following:

- A list of method invocations ($\langle m_1$,`register()`$\rangle, \langle m_2$,`register()`$\rangle$)
- The priority scheduler from Figure 6.3
- The initial value `()` for the instance variable `display`
- The test harness H, giving verdicts at each scheduling decision point

6.4.1 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.

Implementation of the test driver can be done with techniques adapted from the Creol-specific model checker already mentioned in Section 4.5.5. In both cases, what is needed is the ability to control an object's scheduling regime, implementing a scheduler whose behavior is calculated in advance. Controllability is achieved by a *trace* data structure, which contains a sequence of scheduling decisions, with one entry per scheduling point in the model. A special Creol interpreter was implemented that implements this approach.

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 (6.1) represents the states that can reach p, while those states that can avoid p are removed using Formula (6.2). If we don't consider $c_{\neg p}$ in Formula (6.3), 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 (6.1) and (6.2):

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

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.

6.5 Related Work

With the growing dependency on distributed systems and the arrival of multi-core 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 [62] 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 chapter. Schönborn and Kyas [74] 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., [66, 36]), 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., [89]). 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 chapter is to test properties of a program under a specific scheduling regime.

Stone [77] 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. [33] 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 [19] 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 chapter do not help in the actual generation of the test case.

Jasper et al. [56] 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, [88] 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 [82], 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.

6.6 Conclusion and Future Work

This chapter 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 chapter, 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.

Chapter 7

Resource Modeling for Timed Creol Models

This chapter describes the semantics of a timed, resource-constrained extension of the Creol modeling language. Creol is an object-oriented modeling language with a design that is suited for modeling distributed systems. However, the computation model of Creol assumes infinite memory and infinite parallelism within an object. This chapter describes a way to extend Creol with a notion of *resource constraints* and a way to quantitatively assess the effects of introducing resource constraints on a given model. We discuss possible semantics of message delivery under resource constraints, their implementation and their impact on the model. The method is illustrated with a case study modeling a biomedical sensor network.

(The publication which forms the basis of this chapter, "Resource Modeling for Timed Creol Models" [73], was written with Bernhard K. Aichernig, Andreas Griesmayer and Marcel Kyas)

7.1 Introduction

Modeling is an important activity in the design phase of a software project. A formal model can be used to answer questions about a system's functionality, behavior and properties during the specification and implementation phase. By nature, a model focuses on specific aspects of the system under development. For reasons of simplicity and clarity, other aspects of the eventual implementation are abstracted away in the model, e.g. power, bandwidth and memory requirements of components of the system. However, these aspects can be of high importance, for example in embedded systems – models that include these aspects are needed for validating the design against the constraints of the deployment platform.

This chapter describes an enhancement of the modeling language Creol [57] supporting the modeling of resource constraints, specifically restrictions on parallelism, call stack depth, memory consumption, bandwidth and power consumption. Creol is an object-oriented modeling language with asynchronous communication primitives. As described in Chapter 3, a model in Creol consists of classes and objects; objects can have active and reactive behavior. Conceptu-

Resource type	Class attribute	Method attribute	Resources freed
Parallelism	Max. processes	—	at process end
Recursion depth	Maximal depth	—	at process end
Memory	Total memory	Memory needed	at process end
Bandwidth	Avail. bandwidth	Invocation cost	at clock tick
Power	Available power	Execution cost	never

Table 7.1: Types of Resource modeling feasible with the approach described in this chapter. Resources are claimed at process creation time and relinquished depending on the semantics of the modelled resource.

ally, each Creol object has its own processor and handles concurrency independently of other objects. Objects communicate solely via messages and control flow never leaves an object; instead, when a process issues a method call, the receiving object creates a new process that the calling process can synchronize with. These features of Creol make it very suitable for modeling systems of independent, cooperating agents, such as wireless sensor networks.

The idea of the approach is to enhance an existing functional model with information about planned resource usage. This is done by assigning each method a (possibly zero) amount of needed resources, and each class an amount of available resources. This means that resource allocation occurs conceptually at process creation time. The time for relinquishing a resource depends on what the resource models; e.g. in the case of power consumption, the claimed resources are never freed. This abstract concept of resources can be used, among other things, to restrict the amount of parallelism within an object (by giving each method a cost of 1 and the class a number of resources corresponding to the number of allowed concurrent threads), or to model a finite amount of memory or processing power to be claimed by running threads. Table 7.1 shows types of resource that can be modeled with this approach, what the total resources (specified in the class) and the needed resources (specified in the method) mean, and when a resource is freed.

Our definition of "resource" is quite abstract and serves to qualitatively model behavior under resource constraints, for example rendering an object inert after a number of method calls as a model of battery exhaustion. Hence, resource modeling as described herein is not meant to obtain quantitative simulation results of resource depletion.

Various behaviors can be implemented when encountering lack of resources: delaying message delivery, blocking the sender or dropping the message. We give examples for these behaviors, show how to implement them in our rewrite rule-based system and discuss advantages and disadvantages as pertains to modeling.

The rest of this chapter is structured as follows: Section 7.2 describes timed Creol, an extension of the Creol language described in Chapter 3 for modeling timed behavior. Section 7.3 explains how the Creol semantics and interpreter were altered to allow modeling of resource constraints. Section 7.4 presents some results and experiences gained from introducing resource constraints in the BSN case study of Section 3.3.2. Section 7.5 shows how to test against models with resource constraints, and Section 7.6 gives an overview of related work in this area. Section 7.7 concludes the chapter.

7.2 Timed Creol

The base Creol language as described in Chapter 3 does not model time or progress, but there are two extensions implementing timed models of differing expressiveness, described in Kyas and Johnsen [59] and Johnsen et al. [58], respectively.

Both extensions are common and simple: the value of a global clock is accessible to all objects through the expression **now**, which behaves like a read-only global variable of type `Time`. The two approaches differ in the degree of freedom that the model can exhibit in timed constraints. In both approaches, values of type `Time` can be stored in variables and compared with other `Time` values. There is no absolute notion of time; progress can be expressed by adding `Duration` values to observations to obtain other values of type `Time`. An advantage of this design is that specifications in timed Creol are *shift invariant*: properties involving time hold regardless of the point in (absolute) time at which the evaluation happens. (This time extension is inspired by the time model of the Ada programming language [80, Appendix D.8].)

In contrast to the Ada programming language, Creol focuses on modeling and not on implementations. As such, a Creol model is a logical description and certain aspects like preemption due to interrupts are ignored. Interrupt handlers may be modeled by methods of singleton objects, which are invoked as a result of an interrupt signal. Thus, the method described in this chapter allows to model the effect of interrupts without the need of taking the actual machine into account.

In both mentioned approaches to modeling time in Creol, expressing a time invariant looks as follows:

```
1  var t: Time := now;
2  SL
3  await now >= t + 10;
```

The `await` statement in Line 3 guarantees that after evaluating the statement list *SL*, at least 10 units of time pass before the process can continue. (If the effects of *SL* should be *visible* only after 10 time units, then the `await` statement should be placed before *SL*.) Note that the model does not include explicit advancement of time, such as with a *tick* rule; instead, the time is advanced by the interpreter once all object activity has finished in the current tick.

The semantics of Creol allows a process to be suspended indefinitely inside an **await** statement. To ensure forward progress of the system, Kyas and Johnsen [59] introduce the **posit** statement:

```
1  var t: Time := now;
2  SL
3  posit now <= t + 15;
```

Here, Line 3 guarantees that evaluating *SL* takes *at most* 15 time units. A **posit** statement expresses a global property of the system and may result in a system that has no behavior at all (i.e. a system that has no traces); all **posit** statements are proof obligations on the statement level. For details and exact semantics of this version of timed Creol, we once again refer to [59].

The alternative approach of Johnsen et al. [58] implements a different, simpler version of timed Creol, with an operational instead of denotational seman-

tics. The **posit** statement is elided, hence only minimum execution durations can be modeled via **await**. The advantage of this simpler approach is that both implementation and use of the timed constructs is easier; the obvious disadvantage is that progress is not guaranteed.

The results in this chapter were obtained with a timed Creol interpreter written by the author that implements the simpler approach, with as-early-as-possible, run-to-completion semantics. While an **await** statement containing a timed expression could in principle wait forever, the interpreter as implemented guarantees that all enabled processes execute before the global clock is advanced.

7.3 Implementing Resource Constraints

The execution semantics of Creol assumes that an object can execute an arbitrary number of processes. Especially for small embedded systems, this assumption does not hold. It is therefore desirable to be able to model the operating constraints of real systems in Creol. This section presents the implementation approach that was taken to adapt the semantics and execution engine of Creol to deal with resource constraints.

Creol models can be animated on the Maude rewrite engine. In Maude, the execution state of the model is represented as a *state* containing terms representing Creol objects, classes and pending method invocations. The representation of an object `O` of class `C` is

```
< O : C | Att: AL, Pr: { BL | SL}, PrQ: PL >
```

`O`, `C`, `AL`, `BL`, `SL`, and `PL` are typed variables, where object `O` is an instance of class `C` with instance variables `AL` and an active process that consists of local variable bindings `BL` and a list of statements `SL`. The list of pending processes is represented by `PL`. Classes and method invocations have a similar Maude notation.

To implement resource-constrained objects, the notation for classes was updated to contain an attribute `RLimit`:

```
< C : Class | Inh: I, Param: P, Att: S, Mtds: M, RLimit: N >
```

The rest of the attributes are standard and are used for inheritance (`Inh`), constructor parameters (`Param`), Attributes (`Att`) and methods (`Mtds`). The new `RLimit` attribute tells how much memory / processing capacity objects of this class can supply to their processes.

Similarly, an additional method definition was introduced that specifies, in addition to the method name `M`, parameters `P`, local variables `A` and code `C`, how much resources a method needs when called:

```
< M : Method | Param: P, Att: A, Code: C, RNeed: N >
```

With $limit(O)$ the resource limit of an object O (as determined by its class), $P(O)$ the object's set of active processes, and $cost(P)$ the cost of a process (or

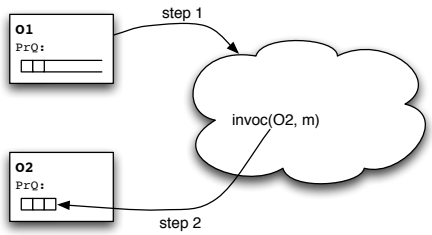

Figure 7.1: Method calls in Creol. For restricted objects, the caller can be blocked (delay in step 1), the call can be delayed infinitely "in the cloud" (delay in step 2) or the call can be dropped (only step 1 happens and the `invoc` term is discarded).

zero if the process has no resource cost), the following invariant needs to hold for all constrained objects:

$$\sum_{p \in P(O)} cost(p) \leq limit(O)$$

The *dynamic semantics* of Creol is given as a set of Maude *rewrite rules* operating on parts of this state. When a rule of the form `<C1> => <C2>` is executed, the part of the state matching `<C1>` is replaced by `<C2>`. For example, the rewrite rule for the `skip` statement of Creol looks as follows:

```
  < O : C | Att: AL, Pr: { BL | skip ; SL}, PrQ: PL >
=>
  < O : C | Att: AL, Pr: { BL | SL}, PrQ: PL >
```

The left-hand side of this rule matches any object with an active process having **`skip`** as its next statement. Such an object is replaced with an object identical in every way except that the **`skip`** statement is removed and the remaining statement list **`SL`** left for execution. Rules for other statements follow the same pattern, but typically have more effect, such as rebinding variables, creating, destroying and scheduling processes or creating new objects.

7.3.1 Possible Semantics of Message Delivery

Delivering a message to an unconstrained object, or to an object that has enough free resources, always succeeds. A new process is created and will be scheduled by the object in due time. However, when the object cannot accept the message and create a process, various behaviors are possible:

1. The message delivery can be *delayed* until the callee can accept it, without the caller being blocked.

2. The caller can be *blocked* until the callee can accept the message.

3. The message can be *dropped*; if the callee cannot accept it, the message is lost.

```
 < C : Class | Inh: I, Param: AL, Att: S1, Mtds: MS, RLimit: N >
 < O : C | Att: S, Pr: P, PrQ: PL >
 invoc(O, m, param)
=>
 < C : Class | Inh: I, Param: AL, Att: S1, Mtds: MS, RLimit: N >
 < O : C | Att: S, Pr: P, PrQ: (PL, createProcess(m, param) >
if nResources(P, PL, m) < N and idleOrSelfcall(P)
```

Figure 7.2: The (slightly simplified) conditional rewrite rule for creating a new process `m` in a constrained object `O`. A process is created if there are enough resources available and if the object can accept a method invocation (i.e., is idle or its current process issued the call).

```
 < O' : C' | Att: S', Pr: { BL | call(O, m, param) ; SL' },
             PrQ: PL' >
 < C : Class | Inh: I, Param: AL, Att: S1, Mtds: MS, RLimit: N >
 < O : C | Att: S, Pr: P, PrQ: PL >
=>
 < O' : C' | Att: S', Pr: { BL | SL' }, PrQ: PL' >
 < C : Class | Inh: I, Param: AL, Att: S1, Mtds: MS, RLimit: N >
 < O : C | Att: S, Pr: P, PrQ: PL >
 invoc(O, m, param)
if nResources(P, PL, m) < N
```

Figure 7.3: The (slightly simplified) conditional rewrite rule for invoking a method of object `O` from object `O1`. Evaluation of the rule is delayed until `O` is in a position to create a process `m`.

Figure 7.2 shows a simplified version of the rule for creating a new process in an instance of a restricted class. A new process is created only when adding it to the object's process queue does not exceed the available resources. This rule implements delayed message delivery.

To implement a delay of the message sender, another rule has to be added that is shown in Figure 7.3. This rule blocks the sender until the receiver can accept the message.

To implement message loss in a Creol model, yet another rule has to be added; a simplified version is shown in Figure 7.4. This rule works in concert with the process creation rule of Figure 7.2; the `[owise]` Maude attribute guarantees that the invocation is only dropped if the process cannot be created.

All of the possible behaviors of message delivery are meaningful in some context. Dropping messages comes closest to the behavior of a system of loosely-coupled components, such as a network of wireless sensors or the datagram level in a TCP/IP network. On the other hand, this model behavior requires extensive changes of the Creol model, compared to an unconstrained model, that are not necessary for the other two possible behaviors. Specifically, every method call that expects a return value has to be implemented with a timeout and an error path:

```
1  var t: Time := now; var l: Label[Int]; var result: Int;
```

```
  < C : Class | Inh: I, Param: AL, Att: S1, Mtds: MS, RLimit: N >
  < O : C | Att: S, Pr: P, PrQ: PL >
  invoc(O, m, param)
=>
  < C : Class | Inh: I, Param: AL, Att: S1, Mtds: MS, RLimit: N >
  < O : C | Att: S, Pr: P, PrQ: PL >
[owise]
```

Figure 7.4: The rewrite rule implementing message loss, working in concert with the process creation rule of Figure 7.2. The left-hand sides of both rules are identical, but this rule only applies if no other rule matches the left-hand side (via Maude's `[owise]` attribute).

```
2  var success: Bool;
3
4  l!o.m();
5  await l?; l?(result); success := true
6  []
7  await now >= t + 10; success := false;
8  if (success)
9    ... // use 'result' here
10 else
11   ... // recover from timeout here
12 end
```

Delaying message delivery and suspending the sending process both model reliable message delivery. Delivery delay models a "smart" network, or an application-transparent buffer-and-resend layer of the sender. Blocking the sender models a tightly-coupled system, probably implemented on a single machine, where querying the receiver's state does not incur sending a message.

7.3.2 Possible Semantics of Resource Allocation and Deallocation

Resources, as understood in this chapter, are always claimed by processes, and are necessary for a process to run. A class specifies how many of these abstract resources its instances have available, and how many (possibly zero) of these resources are claimed by processes running the different methods.

In all cases, resources are claimed upon process start, but different times of freeing resources influence what the resource models.

1. If resources are freed upon process end, the resource puts an upper limit on the *number of processes that can run simultaneously*. This limit on the number of active processes models memory usage or restricted parallelism / recursion depth, depending on the intents of the modeler.

2. If resources are freed after an amount of time has passed in the simulation, the resource puts an upper limit on the *number of method calls per time unit*. This resource limit models bandwidth, i.e. limited communication between objects in the model.

3. Resources that are never freed put a limit on the *total number of method calls* that can run during a simulation of the model, i.e. the objects "run out of power".

Resource constraints, as described in this section, can be used to model and validate a variety of behavior:

Restricted parallelism : To model an object that has restricted parallelism, assign each method a cost of 1 and the class a limit corresponding to the maximum number of running processes.

Recursion depth : To validate that a model run does not exceed a certain depth of self-calls, assign all involved methods a cost of 1 and the class a limit corresponding to the maximum recursion depth. If the methods contain release points, outside calls (that can be used to model interrupts) also factor in the maximum depth.

Memory consumption : Assign the class the amount of memory that is available, and each method its memory cost. This assumes that an object models a physical processor, for example a sensor node.

Bandwidth : Assign the class an amount of bandwidth that is available for method calls in each cycle, and each method its invocation cost. Processes can be created as long as bandwidth is still available; bandwidth again becomes available in the next cycle. Simulating bandwidth-constrained models needs a timed interpreter.

Power consumption : Assign the class an amount of power that is available to processes, and each method its power cost. Power is only ever decreased; an object becomes unresponsive when its power is used up. Similar to memory consumption, the power cost of a method should be a worst-case estimate.

7.4 Modeling with Resource Constraints

This section describes the experiences gained when adding resource constraints to the BSN case study (see Section 3.3.2). The motivation was to model collision of messages, which was not considered in the case study originally. Section 7.4.1 shows the results of a qualitative model, which uncovered a case of unintentional tight coupling between components in the case study, while Section 7.4.2 shows quantitative results of the influence of varying bandwidth constraints and network topologies on the timed behavior of the sensor network.

7.4.1 Results of Adding Restricted Parallelism

The classes `SensorNode` (Figure 3.9) and `Network` (Figure 3.10), together with a class `SinkNode` (not shown) implement a simple flooding routing protocol. In the original case study, *network collisions* were not considered – an arbitrary number of nodes were allowed to broadcast data at the same time.

An obvious way to model the incremental-backoff strategy of resending packets on collision is to restrict the network to only one `broadcast` method at a

Bandwidth	Topology 1 (mixed)	Topology 2 (linear)	Topology 3 (star)
4	14	14	2
3	15	19	3
2	22	29	5
1	45	59	11

Table 7.2: Time for 3 messages each from 4 sensors to arrive at the sink node, depending on topology and available bandwidth.

time. This is very straightforward using the presented resource limit framework – simply assign a cost of 1 to the `broadcast` method and a single resource to the class, while delaying message delivery but allowing the sender to continue running (with these semantics, the timeout-and-resend behavior is implicit, allowing the original model's code to stay in place). With these constraints in place, the original model deadlocked.

The cause of the deadlock was identified in Line 14. The original model had a synchronous call to the receiving node's `receive` method at that point. This serialized message delivery, forcing the receiving node to finish processing the message (including a recursive call to `Network.broadcast`) before the next node would even receive the message. Converting the synchronous call to an asynchronous call allowed the model to run to completion.

While the functional aspects of the model were correct (all messages arrived at the sink node during a simulated run), constraining the `Network` class uncovered what is arguably a modeling error – since that class models the behavior of the "air space" between nodes, messages broadcast by one node should reach all of that node's neighbors at the same time, which must be modeled via asynchronous calls. Constraining the network to its intended behavior helped uncover this modeling error – before, one message would reach the sink and be "on the air" multiple times, rebroadcast from different nodes, before the first broadcast had finished.

7.4.2 Results of Modeling Bandwidth

We now show some simulation results using bandwidth modeling on the timed Creol interpreter. The resource modeled was bandwidth of the `Network` object (see Figure 3.10, page 30), specifically the `broadcast` method which gets called by the sensor nodes to send a message. The `SinkNode` object was modified to record the time of each incoming message; each `SensorNode` object was set up to send three messages. To add another variable, two topologies were evaluated in addition to the "mixed" topology of Figure 3.8. We chose the most extreme topologies for a network of 4 nodes: one where each node is directly connected to the sink node (the "star" topology, which is expected to result in minimal time) and one where the nodes are arranged in a linear fashion, with the last sensor node connected to the sink (the "linear" topology, which should result in maximal time).

Table 7.2 shows the measurements obtained by animating the resource-constrained model on the timed Creol interpreter. The available bandwidth was varied between 1 and 4 messages per timer tick – an available bandwidth of zero results in no arrived messages and hence infinite time to completion, and

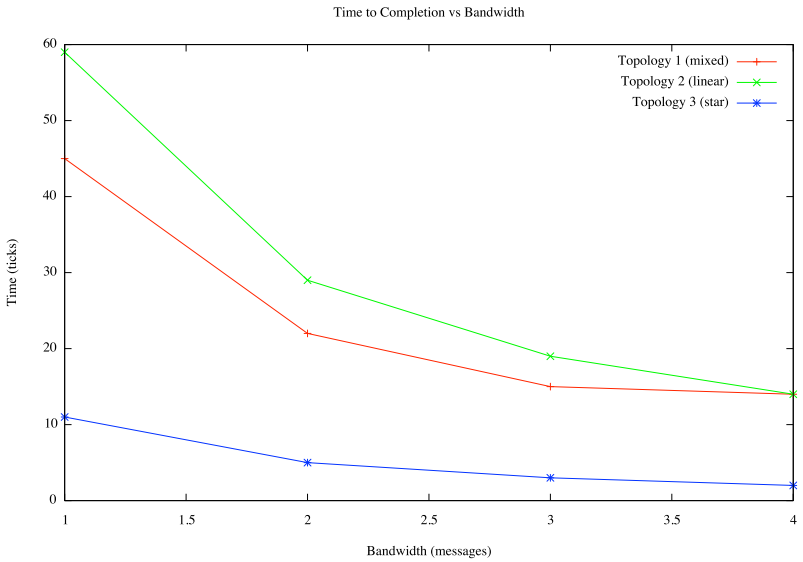

Figure 7.5: Arrival time of the last of 12 messages from 4 sensors, depending on topology and available bandwidth.

a bandwidth of more than 4 messages does not alter the results since there are only four sensor nodes.

Figure 7.5 shows the results in graphical form; as expected, the star topology has significantly lower transmit times than the other two topologies.

7.5 Testing Against Resource Constraints

One of the advantages of a formal model that is executable is that it can serve as *test oracle*, i.e. to validate the implementation (model-based testing). This section explores the possibilities of using a resource-constrained model for testing.

Testing in general means comparing the behavior of a *System under Test* (SuT) against a specification of the system in order to reach a verdict about the conformance of the SuT w.r.t. the specification. A *conformance relation* is used to determine whether observed behavior constitutes allowed behavior with respect to the specification. For abstract datatypes, algebraic specifications can be a good fit for testing [44]. To test the communication of reactive systems with input and output behavior, labeled transition systems [83] and CSP [15] have been employed to good effect.

In our case, the models are written in Creol. Following past work [1], we use the following points to arrive at a test methodology:

- Our *test assumption* is that model and implementation have similar structure and hence equivalent paths of execution.
- To prepare for testing, the tester instruments both model and implementation to record the flow of control at certain meaningful points (e.g. in

the case study at the points where nodes send and receive messages).

- Both Model M and implementation I can thus be seen as mappings from an initial stimulus st to a set of sequences of observed events:

$$st_M \xrightarrow{M} (events_M) * \ ; \ st_I \xrightarrow{I} (events_I) *$$

(In the case of a deterministic model or implementation, the cardinality of the set will be 1 because a stimulus will always result in the same sequence of observed events.)

- A conforming implementation may only show behavior that the model can also exhibit:

$$I(st_I) \subseteq M(st_M) \tag{7.1}$$

Equation 7.1 can be used as the basis of testing an implementation, both against the constrained and the unconstrained model. That equation is not an equality since it is useful for the model to be less restrictive (i.e. more abstract) than the implementation.

7.5.1 Calculating Test Inputs

In [47], the authors present a way of using *dynamic symbolic (concolic) execution* to calculate a set of inputs for a given Creol model that maximizes model path coverage. Briefly summarizing, the approach consists of remembering the symbolic conditions influencing each branching point and scheduling decision of the model, and using a constraint solver to find input values that force the model into another branch of execution during the next run. Each iteration gives a new stimulus st that results in different parts of the model being executed. A full set of these test inputs comprises a test suite with full (bounded) path coverage of the model. In other words, the calculated simuli give us equivalence classes of traces of the model.

The approach has been implemented using a customized Creol interpreter and the Yices constraint solver [90]. For further details, we refer to [47].

7.5.2 Validating a Recorded Trace Against the Model

The traces $I(st_I)$ of equation 7.1 are obtained by animating the SuT with an initial stimulus st, which is the test input, and recording the events in the implementation. In the case of a non-deterministic implementation, approaches similar as the one described in [32] can be employed to increase the variety of traces obtained from one test input. The stimuli st are calculated from the model, as described in Section 7.5.1, and translated into inputs of the SuT by a suitable adapter function.

Recording traces from the SuT is done by instrumenting the source code or by observing events by other means, for example recording network packets. While we have not yet recorded the activity of a physical network of biomedical sensors, we used the approach described in this section on a large conventional software system (see [1]). In that case, recording traces of events was done by instrumenting the system via aspect-oriented programming.

Instrumented model:

```
1  ...
2  with Network // receive data from outside
3    op receive(in data: [Int,Int]) ==
4      if ~(data in received) then
5        await tester.request("Receiving";);
6        queue(data;);
7        received := received |- data
8      end
9  ...
```

Tester:

```
1  ...
2  allow("Receiving";);
3  await pendingReceiving = 0;
4  ...
```

Figure 7.6: Instrumented model and tester. The tester consists of a sequence of `allow` / `await` calls that replay the sequence of events observed on the implementation. The model blocks at the instrumented points (e.g. at Line 5) to synchronize with the tester.

Validating the recorded implementation traces against the model (i.e. checking that the model can reproduce the sequence of events, as per equation 7.1) is done by generating a *tester* that runs in parallel with the model, restricting the model's nondeterminism and forcing it to either exhibit the same trace of events, or deadlock. This post-hoc validation is in contrast to many other approaches [15], which execute model and SuT in parallel. One consequence of the approach is that there is no distinction between input and output on the implementation trace level, but in [1] we describe how to generate a tester that simulates the model according to specific implementation events that represent user input or environment action.

Figure 7.6 shows the `receive` method of the model of the `Node` class of Figure 3.9, instrumented for testing purposes (Line 5), and a fragment of the tester that allows one "Receiving" event to happen. The tester is essentially a sequence of these `allow` / `await` calls, forcing the model to emit the specified events in the given order. The authors implemented a tool, further described in [2], to automatically generate the tester and supporting code from a recorded trace.

7.5.3 Obtaining Test Verdicts

Traces recorded with the calculated stimuli as described in Section 7.5.1 can be used to validate an implementation, as described in Section 7.5.2. This section discusses how to reach a test verdict.

- If the *constrained* model cannot run to completion from a stimulus st calculated from the *unconstrained* model, this means the constraints as specified are too restrictive. This is not a test failure as per Equation (7.1), but

it needs to be investigated by the modeler whether the model is expected to fail with the given input and constraints or not.

- If the *unconstrained* model cannot replay an implementation trace (the tester deadlocks), the implementation does not conform to the model; the test verdict is *Fail*[1].

- If the *constrained* model cannot replay an implementation trace (but the unconstrained model can), the implementation does not conform to the constrained model. This might indicate an implementation error or too restrictive a constraint model; in any case, the test verdict is *Fail*.

- If both models can replay the implementation trace, the test verdict is *Pass*.

7.6 Related Work

Modeling bounded computing resources is relevant because micro-controllers expose the programmer to a bounded call depth, either explicitly or because of memory constraints. For example, the PIC family of micro-controllers has an explicit maximal call depth between 2 and 31, depending on the model. Version 2 of TinyOS [61], an operating system for wireless sensor networks, contains `tos-ramsize`, a tool for *static stack depth analysis* calculating worst-case memory usage by summing stack usage at call points for the longest path through the call graph, and adding stack usage of all interrupt handlers. The theory behind this tool is presented in [70]. Being a simple tool, `tos-ramsize` does not handle recursion. McCartney and Sridhar [63] present `stack-estimator`, a similar tool for the TinyThread library. The value of our work is that resource constraints can be expressed already on the modeling level, and that the model can be validated by simulation. Also, we present a unified approach to modeling call stack depth and restricted parallelism in a model.

Foster et al. [40] make a strong case for checking a model under resource constraints via an example deadlocks in a proven-deadlock-free web service deployment. These deadlocks arose because of thread starvation – the proof of deadlock-freedom did not take the maximum number of threads of the underlying implementation into account. In their approach, the underlying BPEL (Business Process Execution Language) web service orchestration and the thread pool of the system that executes the service requests are modeled together as a labeled transition system. Model checking is then used to ascertain deadlock freedom under resource constraints. Another extensive work using automata to model resource consumption is Chakrabarti et al. [16], where interface automata are used to express the behavior and resource consumption of components, and a compositional game approach is used to calculate the behavior and resource consumption of a composition of components. Chothia and Kleijn [20] present Q-Automata, an automata-based high-level view of components enriched with

[1] In case of non-deterministic models, a backtracking interpreter has to be used to reach a definitive verdict.

costs for taking transitions, with an algebraic notion of combining resources sequentially and in parallel which can model bandwidth, time, power consumption etc. via different operators in the Q-algebra for different resource types.

Our work deals with modeling systems on a lower level of abstraction than using automata models, using Creol [57], an imperative, object-oriented modeling language with asynchronous communication between objects. Similar work was done by Verhoef et al. [87], who use the timed variant of the modeling language VDM++ to model distributed embedded systems. They model processing time, schedulability and bandwidth resources by enriching timed VDM++ with a notion of CPUs, communication buses and asynchronous communication, and loosening the global time model of standard timed VDM++. Creol supports many of the changes necessary for modeling distributed systems in the core language already. Kyas and Johnsen [59] use Creol to model timing aspects of wireless sensors, but do not consider resource constraints of that platform.

7.7 Conclusion

This chapter presented a flexible way of adding resource constraints to a behavioral model written in Creol. The original motivation was to model collision of messages sent over a wireless sensor network, but the approach proved applicable for a whole range of resource constraints: restricted parallelism, recursion depth, memory usage, bandwidth and power consumption. The approach was validated when resource constraints showed that the original model, while showing the expected outcome (messages arriving at the sink node), exhibited too tight coupling between sensor objects and network object and hence did not model the Flooding algorithm correctly.

Adding resource constraints to an existing Creol model requires only one annotation per class and one annotation per constrained method and no required changes to the behavioral specification; if needed, timeout behavior can be added to the model in a simple, straightforward way using the nondeterministic statement of Creol. We believe that the approach is easily adaptable for VDM++ and similar modeling languages. The value of our approach lies in the ease in which it can be added to an existing Creol model, and in the way different behaviors of message delivery can be explored with only local changes in a model. It should be noted that the results obtained by simulating the model are sound but not necessarily complete – while the presence of deadlocks caused by resource constraints can be shown, their absence can only be proven by model-checking, which restricts the size of the model. Nevertheless, experience has shown that the approach can give valuable insight into the behavior of a system that is confronted with limited computing resources.

Chapter 8

Conclusion

This thesis explored the multiple ways in which existing, executable models of the functionality of software or hardware/software systems can be used: for testing, documentation, description of system behavior, and simulation using multiple interpreters with different operational semantics. Speaking personally for a moment, upon finishing this work I see modeling much more as a process of collecting data about a system in a structured form, and much less as trying to specify its behavior (or, trying to program it in a non-executable language)[1] – the characteristic of data being, of course, that it can be used in multiple, sometimes surprising ways.

8.1 On the Multiple Uses of Formal Models

It can be seductive to see a model, especially one written in a language with an operational semantics like Creol, as some sort of "poor man's program". To counter this view, here are the ways that the existing case studies of the Credo project were put to use in the course of this thesis:

- For describing and modeling the functionality of a software or hardware/-software system (the original reason for creating them).

  First and foremost, a model can be *executed*, if the modeling language has an operational semantics. The act of observing a model "in action" gives the modeler valuable insights about its behavior, if the model state is visualized properly.

- As test oracles, checking the functionality of the systems they model as a whole (Chapters 4, 5) and of single components (Chapter 6).

  Here, the main contribution of this thesis is describing and implementing a way of adapting an existing model, in a minimally-invasive way (insertion of trace points at chosen locations), to act as a test oracle. The existing toolchain (Creol compiler, interpreter and editing environment) can be used as-is; the test case generator was added to the normal workflow.

- For test input calculation.

[1] I realize this point is obvious to more advanced practitioners.

Test inputs can be derived even from informal descriptions of a system, via derivation of equivalence classes. Chapter 5 shows how to calculate equivalence classes from a Creol model, by running it in an interpreter with concolic execution semantics.

- For modeling, testing and visualizing the operation of the system in adverse circumstances, operating under constrained bandwidth, memory or timing conditions.

  Chapter 7 shows how to simulate the effect of environmental constraints on executable models. The approach mandates no code changes to the models themselves; for added precision, timeout behavior can be modelled (which is not needed in the unconstrained, untimed interpreter that was used during development of the case studies). Again, the approach works by changing the interpreter and adding some annotations to the models.

Most of these uses can be characterized as testing or validation – since a model purports to describe a software or software/hardware system, verdicts, predictions and axioms extracted from the model will necessarily pertain to the system and be useful in relation to it.

8.2 Using Models As Data: Tools Created

A more pragmatic result of this thesis was the creation of tools for working with the models. What underlines the data nature of the models was the number of interpreters that was created, utilizing each model for different purposes:

- A timed version of the interpreter, following the semantics of Johnsen et al. [58].
- An interpreter that allows to both record and control the scheduling decisions of Creol processes.
- Interpreters implementing the semantics of restricted parallelism, memory and bandwidth (an interpreter implementing restricted power was not implemented but is a trivial adaptation of the existing ones).
- An implementation of dynamic symbolic execution (courtesy of Andreas Griesmayer).

All the interpreters are variations of the original interpreter as contained in the Creol tools package [24], courtesy of Marcel Kyas.

Additionally, a specialized model checker, a test case generator and an editing environment for Creol on the Emacs editor were created by the author, who also took over maintenance of the Eclipse plugin from Johan Dovland.

8.3 Experiences with Creol Modeling

During his work on the Credo project, the author was fortunate to work closely with the authors of the case studies described in Chapter 3. The models created during the case studies are the largest known Creol models in existence, so the tools were pushed to limits not previously encountered. The author himself

uncovered a performance problem in the Creol compiler, by compiling a test method consisting of 600 lines of code generated from an ASK system trace. But all in all, the tools and the language held up well; the case studies were completed to the satisfaction of all participants.

Often mentioned as particular advantages of Creol were the easily understandable semantics and the mostly familiar syntax of the language, the possibility to express different inter-object communication patterns (blocking vs. non-blocking, method return values vs. callbacks), and the cooperative concurrency model of Creol with its reduction of accidental race conditions.

A phenomenon seen in the ASK case study was that the model actually got *smaller* during the modeling activity; this was attributed to increased understanding of the system's behavior and the resulting higher abstraction level in the model. The BSN case study model went through a similar process, but stayed at a constant code size, gaining new features instead of shrinking its code base.

8.4 Future Work

At the time of this writing, the author has been hired by the University of Oslo to continue work on the Creol toolchain and Creol language. Hence, some items in this section describing planned future work are already under active development, and most of them will be implemented within the next three years.

Since this thesis has been written, the Creol language has been extended with a *datatype language*; it is planned to adapt the testing approach to use *parameterized events*, i.e. events that carry additional information. This approach will lead to closer correspondence between test case and SuT, and to less spurious test failures.

Similarly, the visualization and editing tools will be extended to not only show the model's *current state* but also its *execution history* in a graphical way. Many times, the modeler needs to know not only what state the model is in, but also how it got there and what its state was in a previous point in time. A good simulation and visualization tool will be an invaluable tool.

Finally, the question of verification and model-checking of Creol models is still not solved to the author's satisfaction. While full, automated state-space exploration of Creol models is clearly infeasible (for starters, Creol has infinite datatypes), its cooperative scheduling makes techniques developed in sequential settings seem within reach. Some preliminary work in that direction seems encouraging.

More generally, the passive testing approach of this thesis can be adapted to other modeling languages; in general, the techniques for enriching functional models to use them in other contexts (testing, domain modeling, simulation of quantitative aspects of the implementation, ...) are sufficiently general that they can be applied to almost any modeling formalism. The idea of adapting formal models for testing purposes is really powerful, and might help to integrate formal modeling into both the specification and testing phase of a software project. Again, the success of this in an industrial setting depends on tool support and documentation, so enough work remains to be done to make the Creol language a success in this area.

Bibliography

[1] B. Aichernig, A. Griesmayer, R. Schlatte, and A. Stam. Modeling and testing multi-threaded asynchronous systems with Creol. *Electronic Notes in Theoretical Computer Science*, 243:3–14, 2009. Proceedings of the 2nd International Workshop on Harnessing Theories for Tool Support in Software (TTSS 2008).

[2] B. K. Aichernig, A. Griesmayer, E. B. Johnsen, R. Schlatte, and A. Stam. Conformance testing of distributed concurrent systems with executable designs. In de Boer et al. [25], pages 61–81.

[3] B. K. 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. `http://www.iist.unu.edu/www/docs/techreports/reports/report418.pdf`.

[4] Almende website. `http://www.almende.com`. Last accessed December 2, 2009.

[5] J. H. Andrews and Y. Zhang. General test result checking with log file analysis. *IEEE Trans. on Software Engineering*, 29(7):634–648, 2003.

[6] F. Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.

[7] ASK community systems website. `http://www.ask-cs.com`. Last accessed November 27, 2010.

[8] ACC: The AspeCt-oriented C compiler. `http://www.aspectc.net`. Last accessed December 2, 2009.

[9] H. Barringer, K. Havelund, D. E. Rydeheard, and A. Groce. Rule systems for runtime verification: A short tutorial. In S. Bensalem and D. Peled, editors, *RV*, volume 5779 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2009.

[10] 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, *IWTCS*, volume 147 of *IFIP Conference Proceedings*, pages 179–196. Kluwer, 1999.

[11] A. Bertolino, H. Muccini, and A. Polini. Architectural verification of black-box component-based systems. In N. Guelfi and D. Buchs, editors, *RISE*,

volume 4401 of *Lecture Notes in Computer Science*, pages 98–113. Springer, 2006.

[12] R. S. Boyer, B. Elspas, and K. N. Levitt. SELECT – A formal system for testing and debugging programs by symbolic execution. *SIGPLAN Notices*, 10(6):234–245, June 1975.

[13] E. Brinksma and J. Tretmans. Testing transition systems: An annotated bibliography. In F. Cassez, C. Jard, B. Rozoy, and M. D. Ryan, editors, *MOVEP*, volume 2067 of *Lecture Notes in Computer Science*, pages 187–195. Springer, 2000.

[14] 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 *Lecture Notes in Computer Science*, pages 542–547. Springer, 2005.

[15] A. Cavalcanti and M.-C. Gaudel. Testing for refinement in CSP. In M. Butler, M. G. Hinchey, and M. M. Larrondo-Petrie, editors, *9th International Conference on Formal Engineering Methods (ICFEM) 2007*, volume 4789 of *Lecture Notes in Computer Science*, pages 151–170. Springer, 2007.

[16] A. Chakrabarti, L. de Alfaro, T. A. Henzinger, and M. Stoelinga. Resource interfaces. In R. Alur and I. Lee, editors, *Embedded Software*, volume 2855 of *Lecture Notes in Computer Science*, pages 117–133. Springer, 2003.

[17] F. Chang and J. Ren. Validating system properties exhibited in execution traces. In R. E. K. Stirewalt, A. Egyed, and B. F. 0002, editors, *ASE*, pages 517–520. ACM, 2007.

[18] F. Chen and G. Rosu. Mop: an efficient and generic runtime verification framework. In R. P. Gabriel, D. F. Bacon, C. V. Lopes, and G. L. S. Jr., editors, *OOPSLA*, pages 569–588. ACM, 2007.

[19] J.-D. Choi and A. Zeller. Isolating failure-inducing thread schedules. In *International Symposium on Software Testing and Analysis*, pages 210–220. ACM, 2002.

[20] T. Chothia and J. Kleijn. Q-Automata: Modelling the resource usage of concurrent components. *Electronic Notes in Theoretical Computer Science*, 175(2):153–167, 2007.

[21] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[22] 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.

[23] Credo: Modeling and analysis of evolutionary structures for distributed services (IST-33826). `http://www.cwi.nl/CREDO`. Last accessed March 14, 2010.

[24] Creol Tools. Creol compiler and interpreter for Unix-like systems. `http://folk.uio.no/kyas/creoltools/index.html`. Last accessed November 20, 2009.

[25] F. S. de Boer, M. M. Bonsangue, and E. Madelain, editors. *Formal Methods for Components and Objects, 7th International Symposium, FMCO 2008, Sophia Antipolis, France, October 21-23, 2008, Revised Lectures*, volume 5751 of *Lecture Notes in Computer Science*. Springer, 2009.

[26] 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 *Lecture Notes in Computer Science*, pages 316–330. Springer, Mar. 2007.

[27] 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 *Lecture Notes in Computer Science*, pages 81–94. Springer, Aug. 2006.

[28] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[29] W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge University Press, 1998.

[30] J. Dovland, E. B. Johnsen, and O. Owe. Verification of concurrent objects with asynchronous method calls. In *SwSTE*, pages 141–150. IEEE Computer Society, 2005.

[31] J. Dovland, E. B. Johnsen, and O. Owe. Observable behavior of dynamic systems: Component reasoning for concurrent objects. *Electronic Notes in Theoretical Computer Science*, 203(3):19–34, 2008.

[32] 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.

[33] O. Edelstein, E. Farchi, Y. Nir, G. Ratzaby, and S. Ur. Multithreaded Java program test generation. *IBM Systems Journal*, 41(1):111–125, Feb. 2002.

[34] J.-C. Fernandez, C. Jard, T. Jéron, 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 *Lecture Notes in Computer Science*, pages 348–359. Springer, 1996.

[35] 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.

[36] E. Fersman, P. Krcál, P. Pettersson, and W. Yi. Task automata: Schedulability, decidability and undecidability. *Information and Computation*, 205(8):1149–1172, 2007.

[37] E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. Schedulability analysis of fixed-priority systems using timed automata. *Theor. Comput. Sci.*, 354(2):301–317, 2006.

[38] J. S. Fitzgerald, A. E. Haxthausen, and H. Yenigün, editors. *Theoretical Aspects of Computing - ICTAC 2008, 5th International Colloquium, Istanbul, Turkey, September 1-3, 2008. Proceedings*, volume 5160 of *Lecture Notes in Computer Science*. Springer, 2008.

[39] J. S. Fitzgerald and P. G. Larsen. Balancing insight and effort: The industrial uptake of formal methods. In C. B. Jones, Z. Liu, and J. Woodcock, editors, *Formal Methods and Hybrid Real-Time Systems*, volume 4700 of *Lecture Notes in Computer Science*, pages 237–254. Springer, 2007.

[40] H. Foster, W. Emmerich, J. Kramer, J. Magee, D. S. Rosenblum, and S. Uchitel. Model checking service compositions under resource constraints. In I. Crnkovic and A. Bertolino, editors, *ESEC/SIGSOFT FSE*, pages 225–234. ACM, 2007.

[41] G. Fraser, F. Wotawa, and P. Ammann. Issues in using model checkers for test case generation. *Journal of Systems and Software*, 82(9):1403–1418, 2009.

[42] G. Fraser, F. Wotawa, and P. Ammann. Testing with model checkers: A survey. *Software Testing, Verification and Reliability*, 19(3):215–261, 2009.

[43] 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.

[44] M.-C. Gaudel and P. L. Gall. Testing data types implementations from algebraic specifications. In R. M. Hierons, J. P. Bowen, and M. Harman, editors, *Formal Methods and Testing 2008*, volume 4949 of *Lecture Notes in Computer Science*, pages 209–239. Springer, 2008.

[45] 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.

[46] I. Grabe, M. M. Jaghoori, B. K. Aichernig, C. Baier, T. Blechmann, F. de Boer, A. Griesmayer, E. B. Johnsen, J. Kleijn, S. Klüppelholz, M. Kyas, W. Leister, R. Schlatte, A. Stam, M. Steffen, S. Tschirner, X. Liang, and W. Yi. Credo methodology. modeling and analyzing a peer-to-peer system in Credo. *Electronic Notes in Theoretical Computer Science*, 2009. 3rd International Workshop on Harnessing Theories for Tool Support in Software (TTSS 2009). ENTCS, Elsevier, Amsterdam (to appear).

[47] A. Griesmayer, B. K. Aichernig, E. B. Johnsen, and R. Schlatte. Dynamic symbolic execution for testing distributed objects. In C. Dubois, editor, *TAP*, volume 5668 of *Lecture Notes in Computer Science*, pages 105–120. Springer, 2009.

[48] A. Griesmayer, B. K. Aichernig, E. B. Johnsen, and R. Schlatte. Dynamic symbolic execution of distributed concurrent objects. In D. Lee, A. Lopes, and A. Poetzsch-Heffter, editors, *FMOODS/FORTE*, volume 5522 of *Lecture Notes in Computer Science*, pages 225–230. Springer, 2009.

[49] A. Groce, K. Havelund, M. Smith, and H. Barringer. Let's look at the logs: Low-impact runtime verification. `http://www.havelund.com/Publications/logscope09.pdf` (submitted for publication).

[50] R. H. Halstead. MULTILISP: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, Oct. 1985.

[51] 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.

[52] C. Hewitt, P. Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In *IJCAI'73: Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[53] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Lüttgen, A. J. H. Simons, S. A. Vilkomir, M. R. Woodward, and H. Zedan. Using formal specifications to support testing. *ACM Computing Surveys*, 41(2):1–76, 2009.

[54] ILOG Solver 6.0 User's Manual. `http://www.lkn.ei.tum.de/arbeiten/faq/man/ILOG/CONCERT/concert20/pdf/solver60userman.pdf`. Last accessed March 12, 2010.

[55] ISO/IEC 9646-1: Information technology - OSI - Conformance testing methodology and framework - Part 1: General Concepts, 1994.

[56] 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.

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

[58] E. B. Johnsen, O. Owe, J. Bjørk, and M. Kyas. An object-oriented component model for heterogeneous nets. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *FMCO*, volume 5382 of *Lecture Notes in Computer Science*, pages 257–279. Springer, 2007.

[59] M. Kyas and E. B. Johnsen. A real-time extension of Creol for modelling biomedical sensors. In de Boer et al. [25], pages 42–60.

[60] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines – a survey. *Proceedings of the IEEE*, 84(8):1090–1123, August 1996.

[61] P. Levis and D. Gay. *TinyOS Programming*. Cambridge University Press, 2009.

[62] J. Magee and J. Kramer. *Concurrency: State Models & Java Programs*. Wiley, 2nd edition, 2006.

[63] W. P. McCartney and N. Sridhar. Abstractions for safe concurrent programming in networked embedded systems. In A. T. Campbell, P. Bonnet, and J. S. Heidemann, editors, *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems (SenSys)*, pages 167–180. ACM, 2006.

[64] S. Meng and F. Arbab. Web services choreography and orchestration in reo and constraint automata. In Y. Cho, R. L. Wainwright, H. Haddad, S. Y. Shin, and Y. W. Koo, editors, *SAC*, pages 346–353. ACM, 2007.

[65] J. Meseguer. Conditioned rewriting logic as a united model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[66] 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 *Lecture Notes in Computer Science*, pages 493–513. Springer, 2001.

[67] C. E. Perkins and E. M. Belding-Royer. Ad-hoc on-demand distance vector routing. In *WMCSA*, pages 90–100. IEEE Computer Society, 1999.

[68] C. E. Perkins, E. M. Belding-Royer, and S. Das. Ad hoc On-Demand Distance Vector (AODV) Routing. RFC 3561 (Experimental), July 2003. `http://www.ietf.org/rfc/rfc3561.txt`.

[69] 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.

[70] J. Regehr, A. Reid, and K. Webb. Eliminating stack overflow by abstract interpretation. *ACM Transactions on Embedded Computing Systems (TECS)*, 4(4):751–778, Nov. 2005.

[71] 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 *Lecture Notes in Computer Science*, pages 338–357. Springer, 2000.

[72] R. Schlatte, B. K. Aichernig, F. S. de Boer, A. Griesmayer, and E. B. Johnsen. Testing concurrent objects with application-specific schedulers. In Fitzgerald et al. [38], pages 319–333.

[73] R. Schlatte, B. K. Aichernig, A. Griesmayer, and M. Kyas. Resource modeling for timed Creol models. *Electronic Notes in Theoretical Computer Science*, 2009. 3rd International Workshop on Harnessing Theories for Tool Support in Software (TTSS 2009). ENTCS, Elsevier, Amsterdam (to appear).

[74] J. Schönborn and M. Kyas. A theory of bounded fair scheduling. In Fitzgerald et al. [38], pages 334–348.

[75] 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 *Lecture Notes in Computer Science*, page 419. Springer, 2006.

[76] M. Sirjani, M. M. Jaghoori, C. Baier, and F. Arbab. Compositional semantics of an actor-based language using constraint automata. In P. Ciancarini and H. Wiklicky, editors, *COORDINATION*, volume 4038 of *Lecture Notes in Computer Science*, pages 281–297. Springer, 2006.

[77] 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.

[78] Sun Microsystems, Inc. *Solaris Dynamic Tracing Guide*, Sept. 2008. Part No: 817-6223-12. `http://dlc.sun.com/pdf/817-6223/817-6223.pdf`. Last accessed November 27, 2009.

[79] SystemTap. `http://sourceware.org/systemtap/documentation.html`. Last accessed November 27, 2009.

[80] S. T. Taft, R. A. Duff, R. L. Brukardt, E. Plödereder, and P. Leroy. *Ada 2005 Reference Manual. Language and Standard Libraries – International Standard ISO/IEC 8652:1995 (E) with Technical Corrigendum 1 and Amendment 1*, volume 4348 of *Lecture Notes in Computer Science*. Springer, 2006.

[81] 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 *Lecture Notes in Computer Science*, pages 134–153. Springer, 2008.

[82] 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.

[83] 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 *Lecture Notes in Computer Science*, pages 127–146. Springer, 1996.

[84] 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 *Lecture Notes in Computer Science*, pages 1–38. Springer, 2008.

[85] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing approaches. Working Paper 04/2006, Department of Computer Science, The University of Waikato, Apr. 2006. ISSN 1170-487X. `http://www.cs.waikato.ac.nz/pubs/wp/2006/uow-cs-wp-2006-04.pdf`.

[86] 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 *Lecture Notes in Computer Science*, pages 86–100. Springer, 2003.

[87] M. Verhoef, P. G. Larsen, and J. Hooman. Modeling and validating distributed embedded real-time systems with VDM++. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM 2006: Formal Methods, 14th International Symposium on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2006.

[88] 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 *Lecture Notes in Computer Science*, pages 82–95. Springer, 2006.

[89] E. J. Weyuker. Testing component-based software: A cautionary tale. *IEEE Software*, pages 54–59, Sept. 1998.

[90] The YICES SMT Solver. `http://yices.csl.sri.com/tool-paper.pdf`. Last accessed December 2, 2009.