Implementing Conformiq Qtronic

Antti Huima
(antti.huima@conformiq.com)

Conformiq Software Ltd.

Abstract. Conformiq Qtronic[1] is a commercial tool for model driven testing. It derives tests automatically from behavioral system models. These are black-box tests [1] by nature, which means that they depend on the model and the interfaces of the system under test, but not on the internal structure (e.g. source code) of the implementation.

In this essay, which accompanies my invited talk, I survey the nature of Conformiq Qtronic, the main implementation challenges that we have encountered and how we have approached them.

Problem Statement

Conformiq Qtronic is an ongoing attempt to provide an industrially applicable solution to the following technical problem: Create an automatic method that provided an object (a model) M that describes the external behavior of an open system (begin interpreted via well-defined semantics) constructs a strategy to test real-life black-box systems (implementations) I in order to find out if they have the same external behavior as M.

(Now the fact that we have created a company with the mission to solve this technical problem hints that we believe it to be a good idea to *employ* automatic methods of this kind in actual software development processes—which is not *a priori* self-evident. In this essay, however, I will not touch these commercial and methodological issues, but will concentrate on the technical problem only.)

Two notes are due, and the first concerns the *test harness*. It is namely usually so that the model M describes a system with different interfaces and behavior than the real implementation that is being tested (IUT, implementation under test). Virtually always, the model M is somehow described on a higher level of abstraction than the real implementation. As a concrete example, SIP (Session Initiation Protocol) [2] is a packet-oriented protocol that is run over a transport (e.g. UDP [3]) and that has an ASCII encoding: every SIP packet is encoded as a series of ASCII characters. However, some possible model M for testing an SIP implementation could define the logical flow of packets with their main contents but not describe, for instance, their actual encoding. How can this model M be used to test a real SIP implementation that requires ASCII encoded packets over UDP? The answer is that the "abstract" tests generated automatically from M are run through a *test harness* that provides the "low-level details" such as encoding and decoding data and managing the UDP packets.

[1] www.conformiq.com

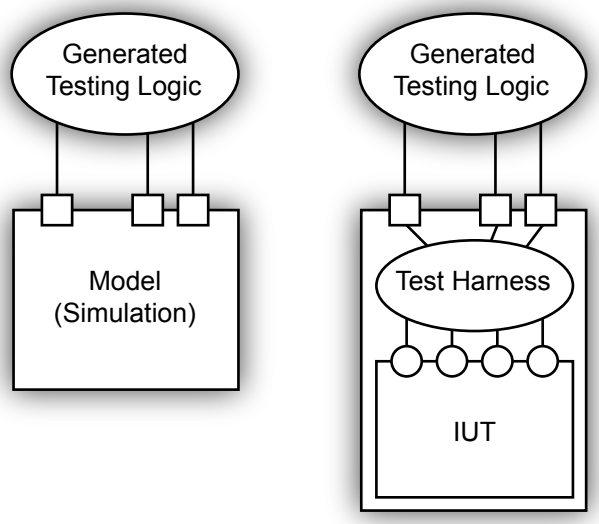

Fig. 1. The relation between a model and a corresponding implementation under test. The test harness bridges the higher-level model with the concrete executable implementation. The logic of the test harness itself is not explicitly tested by the generated testing logic.

What does this mean in terms of testing? The automatic tests derived from M are not testing the codec layer of the SIP implementation *per se*, because the codec was not specified in M. Instead, the tests focus on the "high-level" logic above the codec layers. See Fig. 1.

This is a very commonly occurring pattern. Yet testing the codecs could also benefit from model-based testing. In practical deployments this would be usually carried out by using a different model and different testing set-up that would focus *solely* on the testing of the codecs. Regardless, we have found that succesful deployments of model-based testing usually focus on the "higher level" issues.

The second note concerns the models M in the context of Conformiq Qtronic. For us, the models are—on the internal level—multithreaded Scheme [4] programs that can "talk" with an external, unspecified (open) environment, and the semantics for these programs are given via small-step virtual machine operational semantics [5, 6], defined in the corresponding language specification [7].

Design of Conformiq Qtronic

The development of Conformiq Qtronic has always been driven by two major design goals, which are *industrial applicability* and *openess*. In other words, we

have strived to create a tool that can be applied to real-world problems by "normal" software engineers in the industry, and that can be integrated easily into heterogenous software development tool chains.

The most visible consequence is that Conformiq Qtronic allows the user to use in modeling many constructs that are known to create difficulties for model checking algorithms. Namely, Conformiq Qtronic supports infinite data types (e.g. rational numbers and strings), fully dynamic data with garbage collection, classes with static and dynamic polymorphism, concurrency, and timings. Furthermore, the user is not asked to provide abstractions [8, 9] or slicings [10, 11] of the model, as the design philosophy has been that Conformiq Qtronic should relieve the user of such technicalities.

CHALLENGE:
Legal models can have arbitrarily complex, infinite state spaces

It follows immediately that there exist models that are too difficult for Conformiq Qtronic to handle. Our solution to this problem is that we do not try to solve it, because to provide the tool's *raison d'être*, it is sufficent that there exist *enough* contexts where Conformiq Qtronic can create substantial value.

Another consequence is that many of the known methods for model checking and test generation that assume finite-state specifications [12–20], or even only finitely branching specifications, are of limited value for us. I do not claim that research into model driven testing from finite-state models would be useless in general, especially as there exist case studies that prove otherwise [21]. However, our experience is that the number of industrial systems that have natural finite-state specifications is limited. There exists certainly a larger body of systems that have somewhat natural finite-state *abstractions*, but as I already mentioned, we do not want to force the user to do abstraction manually.

Modeling

The main artifact the user of Conformiq Qtronic must manage is the model, and we believe it is very important that the model can be described in a *powerful* language that is *easy to adopt*.

CHALLENGE:
Providing industrial-strength specification and modeling language

In the out-of-the-box version of Conformiq Qtronic, the behavioral models are combinations of UML statecharts [22, 23] and blocks of an object-oriented programming language that is basically a superset of Java [24] and C# [25]. We have dubbed this semi-graphical language QML (Qtronic Modeling Language). The use of UML statecharts is optional so that models can be described in pure textual notation also.

QML [26] extends Java by adding value-type records (which exist already in C#), true static polymorphism (i.e. templates or generics), discretionary type inference [27–29], and a system for free-form macros (see Fig. 2). To avoid any

```
_rec -> _port ::= _port.send(_rec, -1)
```

Fig. 2. A free-form macro that gives a new syntax for sending data through ports without timeouts. Note that there is no built-in meaning for the arrow literal, and when matched, `_port` and `_rec` can correspond to arbitrary expressions.

misunderstanding it must be mentioned that QML does *not* include the standard libraries that come with Java or C#.

A QML program must specify an open system, i.e. a system with one or more message passing interfaces that are open to the "environment" (please see Fig. 1 again). These interfaces correspond to the testing interfaces of the real system under test. For example, if a QML program starts by sending out a message X, then a conforming system under test must also send out the message X when it starts. Thus, in a very concrete manner, a QML program is an *abstract reference implementation* of the system it specifies. As a matter of fact, a great way to test Conformiq Qtronic itself is to run tests derived from a model against a simulated execution of the very same model (expect no failures!), or mutants [30–32] of it.

In order to clarify this further, we do *not* rely on any existing Java or C# compilers, but have a full custom-built translator for QML. (I elaborate this later in this paper.)

Online and Offline Testing

Conformiq Qtronic offers two complementary ways for deploying the derived tests: *online testing* (*on-the-fly*) and *offline test generation*. Online testing means in our context that Conformiq Qtronic is connected "directly" with the system under test via a DLL (dynamically linked library) plug-in interface. In this mode, the selection and execution of test steps and the validation of the system under test's behavior all take place in parallel. In contrast, *offline test generation* decouples test case design from the execution of tests. In the offline mode, Conformiq Qtronic creates a library of test cases that are exported via an open plug-in interface and that can be deployed later, independent of the Conformiq Qtronic tool.

Conformiq Qtronic supports the testing of systems against *nondeterministic system models*, i.e. again models that allow for multiple different observable behaviors *even against a deterministic testing strategy*—but only in the online mode.

CHALLENGE:
Supporting nondeterministic system models

At the present, the offline test case generator assumes a deterministic system model. One of the reasons for this is that the test cases corresponding to a nondeterministic model resemble trees as at the test generation time the choices that the system under test will make are not yet known. The branching factor of

such trees is difficult to contain, especially in the case of very wide nondeterministic branches (e.g. the system under test chooses a random integer). In contrast, the online algorithm can adapt to the already observed responses of the system under test, and choose the next test steps completely dynamically. [21, 33]

CHALLENGE:
Timed testing in real time

One important source of nondeterminism is time [33–35], especially because the testing setup itself typically creates communication latencies. For example, if the SUT sends out a timeout message after 10 seconds, it can be that the testing harness actually sees the message only after 10.1 seconds due to some slowness in the testing environment. In the same way the inputs to the SUT can get delayed. This is so important that the user interface for Conformiq Qtronic provides a widget for setting the maximum allowed communication latency. This widget actually controls the time bound of a bidirectional queue object that the tool adds implicitly "in front of" the provided system model.

Multilanguage Support

Even though QML is the default modeling language provided with Conformiq Qtronic, the tool supports also other languages. This multi-language support is implemented internally by translating all user-level models into an intermediate process notation. This notation, which we call CQλ, is actually a variant of Scheme [4], a lexically scoped dialect of the LISP language family [36]. For an example, see Fig. 3.

CHALLENGE:
Compiling models into an intermediate language

So, all QML models are translated eventually into LISP programs. Our pipeline for doing this consists of the following components: (1) a model and metamodel [37] loading front-end based on the ECore specification [38] and XMI [39]; (2) an in-memory model repository [40, 41]; (3) a parsing framework for textual languages that supports fully ambiguous grammars [42, 43]; and (4) a graph-rewriting [44–46] based framework for model transformation [40].

First, a QML model consisting of UML statecharts and textual program blocks is loaded into the model repository via the model loading front-end. At the same time, two metamodels are loaded: one for QML and one for CQλ. Originally, the textual program blocks appear in the repository as opaque strings. The next step is to parse them and to replace the strings with the corresponding syntax trees (for which there is support in the QML metamodel). Macro expansion, type checking and type inference happen at this stage. Then the graph rewriter is invoked with a specific *rule set*, which is iterated until a fixpoint; this causes the model to be gradually transformed from an instance of the QML metamodel to an instance of the CQλ metamodel. Eventually, the resulting CQλ model is

```
(define-input-port input)
(define-output-port output)
(define main
  (lambda ()
    (let* ((msg (ref (handshake input #f) 1))
           (_ (handshake (tuple output msg) #f)))
      (main))))
```

Fig. 3. A minimalistic CQλ program that defines an "echo" system: every message sent to the port `input` must be echoed back immediately through the port `output`.

```
replace "Timer trigger" {
} where {
    Transition t;
    TimeoutTrigger trigger;
    t.trigger == trigger;
    t.trigger_cql == nil;
} with {
    t.trigger_cql := `(tuple ,CQL_Symbol("__after__") ,trigger.timeout);
};
```

Fig. 4. A simple graph rewriting rule that generates the CQλ counterpart for a timeout trigger in an UML statechart.

linearized into a textual CQλ program and a fixed CQλ library pasted in. [47, 48] A simple example of a rewriting rule is shown in Fig. 4.

It is important to be able to map the translated CQλ program back to the original user-level model in order to support traceability. There is specific support for this in the CQλ language: program blocks can be linked both lexically as well as dynamically to the user-level model.

Test Generation

So how does Conformiq Qtronic generate tests? The core algorithm is an enumerator for simulated executions of the given model. What makes this difficult is that this enumerator needs to be able to simulate an *open* model, i.e. a model that communicates an environment that has not been specified. Basically the enumerator assumes that a fully nondeterministic environment has been linked with the open model. This creates infinitely wide nondeterministic branches in the state space, because the hypothetical environment could send e.g. any integer whatsover to the system (Conformiq Qtronic supports as a datatype the full set $\mathbb{Z}$). Another set of nondeterministic branches is caused by the internal choices in the model itself, and these can be also infinitely wide (the *model* generates a free integer value). So the trick is how to handle a state space with infinitely many states *and* an infinite branching factor. This we do with *symbolic execution* and I get back to this shortly.

CHALLENGE:
Supporting known testing heuristics

In order to be able to support different testing heuristics, such as transition or state coverage or boundary value analysis [1], Conformiq Qtronic has a built-in capability to include *coverage checkpoints* in the intermediate CQλ-level models. Similar constructs have been called also e.g. coverage items in the literature. [49] A coverage checkpoint is marked in a LISP model by a call to the built-in `checkpoint` procedure. This means that the generation of coverage checkpoints can be fully controlled in the model transformation stage. Indeed, all the various user-level testing heuristics such as transition coverage or boundary value analysis have been implemented in the model transformator via this generic checkpoint facility.

Let us give the set of all possible traces, i.e. sequences of messages, the name $\mathbb{T}$, the set of all coverage checkpoints the name $\mathbb{C}$, and denote the set of booleans by $\mathbb{B}$. The state space enumerator can be seen as an oracle that implements the following functions:

$$\begin{aligned} \mathsf{valid} &: \mathbb{T} \to \mathbb{B} \\ \mathsf{coverage} &: \mathbb{T} \to 2^{\mathbb{C}} \\ \mathsf{plan} &: \mathbb{T} \times 2^{\mathbb{C}} \to \mathbb{T} \end{aligned}$$

The function valid tells whether a given trace is something that the model could produce or not, so it embodies a (bounded) model checker. The next function coverage calculates the set of checkpoints that *must* have been passed on every execution of the model that produces the given trace. Finally, plan calculates an extension (suffix) for a valid trace that can be produced by the model, attempting to find such an extension that it would cause a checkpoint that is not included in the given set to be passed.

Given these oracles, a simplified version of the online mode of Conformiq Qtronic can be described by the following algorithm. Initialize a variable C—which will contain checkpoints—with the empty set. Initialize another variable t—which will contain a trace—with the empty trace. Then repeat *ad infinitum*: If $\mathsf{valid}(t) = \text{false}$, signal 'FAIL' and stop testing. Otherwise, update C to $C \cup \mathsf{coverage}(t)$. Then calculate $t' = \mathsf{plan}(t, C)$. If the next event in t' is an input to the system under test, wait until the time of the event and then send it. Otherwise just wait for some time. In any case, if a message is received from the SUT during waiting, update t accordingly; if not, update t with the message potentially sent, and in any case with the current wallclock reading.

Offline script generation is even easier. Because the model must be deterministic modulo inputs, anything returned by plan works as a test case. The basic idea is to find a set T of traces such that $|T|$ is small and $\bigcup_{t \in T} \mathsf{coverage}(t)$ is large.

In practice, the oracles valid, coverage and others are built around a *symbolic executor* for CQλ. Symbolic execution is well known and has been applied for test

generation and program verification. [50–52] Usually implementing it requires some form of *constraint solving* [53, 54], and so Conformiq Qtronic also sports a constraint solver under the hood. Maybe interestingly, the data domain for our solver is the least D such that

$$\mathbb{Q} \cup \mathbb{B} \cup \mathbb{S} \cup D^0 \cup D^1 \cup \cdots = D$$

where $\mathbb{S}$ denotes the infinite set of *symbols* (opaque, enumerated values). In words, any constraint variable in a constraint problem within Conformiq Qtronic can *a priori* assume a numeric, boolean or symbol value, or a tuple of an arbitrary size containing such values and other tuples recursively. In particular, we present strings (e.g. strings of Unicode [55] characters) as tuples of integers, and value records (`structs`) as tuples containing the values of the fields. This weakly typed structure of the constraint solver reflects the dynamic typing in the CQλ language.

CHALLENGE:
Implementing constraint solver over infinite domains

The constraint solver for Conformiq Qtronic has been developed in-house in C++ because it is tightly integrated with the symbolic executor itself. For example, our symbolic executor has a garbage collector [56] for the LISP heap, and when *symbolic* data gets garbage collected (references to constraint variables), the solver attempts to eliminate the variables from the constraint system by bucket elimination [57]. Handling infinite data domains provides a big challenge, because many of the state-of-the-art methods for solving difficult constraint problems assume finite-domain problems, and we do active research on this area on daily basis.

Scalability

At least in the form implemented in Conformiq Qtronic, model-based testing is a computationally intensive task.

CHALLENGE:
Practical time and space complexity

In practice, both time and memory space are important resource factors for us. Certainly, we work continuously to incrementally improve the time and memory characteristics of Conformiq Qtronic, but we employ also some categorical solutions.

The performance of Conformiq Qtronic (as for most other software applications) becomes unacceptable when it runs out of physical memory and begins swap trashing. To prevent this, Conformiq Qtronic swaps proactively most of the runtime objects—like parts of the symbolic state space—on hard disk. Our architectural solution for this is based on a variant of *reference counting* [56].

One challenge in the near future is to scale Conformiq Qtronic from single-workstation application to *ad hoc* grids [58] or clusters. Simply, we want to

provide the option to run the core algorithms in parallel on all free CPUs in an internal network, such as all the idling Windows and Linux workstations within an office.

Post Scriptum

According to John A. Wheeler,

> We live on an island surrounded by a sea of ignorance. As our island of knowledge grows, so does the shore of our ignorance.

How true is this of model-based testing also! When we started the Conformiq Qtronic journey in early 2003—one hundred years after Wheeler's famous quote—we had a few fundamental questions that we had to solve. Today, our questions have changed in nature but only increased in number! Here are some of those which we have encountered:

—Is it possible to automatically explain the "logic behind" computer-generated test cases to a human engineer?

—Can computer-generated test cases be grouped and catalogued intelligently?

—Is it possible to generate offline test scripts that handle infinite-valued nondeterminism without resorting to full constraint solving during test script execution?

—What forms of precomputation (e.g. forms of abstract interpretation [28, 59, 60]) can be used to reduce the runtime computational burden of online testing?

—Are the better modeling formalisms for system modeling for model-based testing than general modeling languages?

—How can well-researched finite-domain constraint solving techniques (like nonchronological backtracking or conflict set learning) be used to the full extent in the context of infinite-domain problems?

Acknowledgements Conformiq Qtronic research and development has been partially supported by TEKES[2], and ITEA[3], an EUREKA[4] cluster. Special thanks are due to the co-chairs of the conference for their kind invitation.

[2] www.tekes.fi
[3] www.itea-office.org
[4] www.eureka.be

References

[1] Craig, R.D., Jaskiel, S.P.: Systematic Software Testing. Artech House Publishers (2002)

[2] Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., Schooler, E.: SIP: Session initiation protocol. Request for Comments 3261, The Internet Society (2002)

[3] Postel, J.: User datagram protocol. Request for Comments 768, The Internet Society (1980)

[4] Abelson, H., Dybvig, R.K., Haynes, C.T., Rozas, G.J., Iv, N.I.A., Friedman, D.P., Kohlbecker, E., G. L. Steele, J., Bartley, D.H., Halstead, R., Oxley, D., Sussman, G.J., Brooks, G., Hanson, C., Pitman, K.M., Wand, M.: Revised report on the algorithmic language scheme. Higher Order Symbol. Comput. **11**(1) (1998) 7–105

[5] Gunter, C.A.: Semantics of Programming Languages. The MIT Press (1992) ISBN 0-262-07143-6.

[6] Plotkin, G.D.: A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus (1981)

[7] Huima, A. (ed.): CQλ specification. Technical report, Conformiq Software (2003) Available upon request.

[8] Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. ACM Trans. Program. Lang. Syst. **16**(5) (1994) 1512–1542

[9] Ammann, P., Black, P.: Abstracting formal specifications to generate software tests via model checking. In: Proceedings of the 18th Digital Avionics Systems Conference (DASC99). Volume 2., IEEE (1999) 10.A.6

[10] Reps, T., Turnidge, T.: Program specialization via program slicing. In Danvy, O., Glueck, R., Thiemann, P., eds.: Proceedings of the Dagstuhl Seminar on Partial Evaluation, Schloss Dagstuhl, Wadern, Germany, Springer-Verlag, New York, NY (1996) 409–429

[11] Weiser, M.: Program slicing. In: ICSE '81: Proceedings of the 5th international conference on Software engineering, Piscataway, NJ, USA, IEEE Press (1981) 439–449

[12] E. M. Clarke, Jr., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press (2000) ISBN 0-262-03270-8.

[13] Luo, G., Petrenko, A., Bochmann, G.V.: Selecting test sequences for partially-specified nondeterministic finite state machines. Technical Report IRO-864 (1993)

[14] Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines - A survey. In: Proceedings of the IEEE. Volume 84. (1996) 1090–1126

[15] Pyhälä, T., Heljanko, K.: Specification coverage aided test selection. In Lilius, J., Balarin, F., Machado, R.J., eds.: Proceeding of the 3rd International Conference on Application of Concurrency to System Design (ACSD'2003), Guimaraes, Portugal, IEEE Computer Society (2003) 187–195

[16] Tretmans, J.: A formal approach to conformance testing. In: Proc. 6th International Workshop on Protocols Test Systems. Number C-19 in IFIP Transactions (1994) 257–276

[17] Luo, G., von Bochmann, G., Petrenko, A.: Test selection based on communicating nondeterministic finite state machines using a generalized wp-method. IEEE Transactions on Software Engineering **SE-20**(2) (1994) 149–162

[18] Feijs, L., Goga, N., Mauw, S.: Probabilities in the TorX test derivation algorithm. In: Proc. SAM'2000, SDL Forum Society (2000)

[19] Petrenko, A., Yevtushenko, N., Huo, J.L.: Testing transition systems with input and output tester. In: TestCom 2003, Springer-Verlag (2003)
[20] Tretmans, J.: Test generation with inputs, outputs and repetitive quiescence. Software—Concepts and Tools **17**(3) (1996) 103–120
[21] Veanes, M., Campbell, C., Schulte, W., Tillmann, N.: Online testing with model programs. In: ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, New York, NY, USA, ACM Press (2005) 273–282
[22] Object Management Group: Unified Modeling Language: Superstructure. Technical Report formal/2007-02-05 (2007)
[23] Selic, B.: UML 2: a model-driven development tool. IBM Syst. J. **45**(3) (2006) 607–620
[24] Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification. 3rd edn. Prentice Hall (2005)
[25] Michaelis, M.: Essential C# 2.0. Addison-Wesley Professional (2006)
[26] Conformiq Software: Conformiq Qtronic User Manual. Conformiq Software (2007) Publicly available as part of product download.
[27] Milner, R.: A theory of type polymorphism in programming. Journal of Computer and System Science **17**(3) (1978) 348–375
[28] Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer (1999) ISBN 3-540-65410-0.
[29] Pierce, B.C.: Types and Programming Languages. The MIT Press (2002)
[30] Budd, T.A., DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In: POPL '80: Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM Press (1980) 220–233
[31] Offutt, A.J., Lee, S.: An empirical evaluation of weak mutation. IEEE Transactions on Software Engineering **20**(5) (1994) 337–344
[32] Zhu, H., Hall, P., May, J.: Software unit test coverage and adequacy. ACM Computing Surveys **29**(4) (1997) 366–427
[33] Larsen, K.G., Mikucionis, M., Nielsen, B.: Online testing of real-time systems using UPPAAL. In: Formal Approaches to Software Testing. Volume 3395/2005 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2004) 79–94
[34] Bohnenkamp, H., Belinfante, A.: Timed testing with TorX. In: Formal Methods Europe 2005. Volume 3582/2005 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2005) 173–188
[35] Briones, L., Brinksma, E.: Testing real-time multi input-output systems. In: Formal Methods and Software Engineering. Volume 3785/2005 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2005) 264–279
[36] Steele, Jr., G.L., Gabriel, R.P.: The evolution of Lisp. ACM SIGPLAN Notices **28**(3) (1993) 231–270
[37] Object Management Group: Meta Object Facility (MOF) Core Specification. Technical Report formal/06-01-01 (2006)
[38] Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.J.: Eclipse Modeling Framework. 1st edn. Addison-Wesley Professional (2003)
[39] Object Management Group: MOF 2.0/XMI Mapping Specification. Technical Report formal/05-09-01 (2005)
[40] Mellor, S.J., Scott, K., Uhl, A., Weise, D.: MDA Distilled. Addison-Wesley Professional (2004)

[41] Kleppe, A., Warmer, J., Bast, W.: MDA Explained. Addison-Wesley (2003)
[42] Aho, A.V., Johnson, S.C., Ullman, J.D.: Deterministic parsing of ambiguous grammars. Commun. ACM **18**(8) (1975) 441–452
[43] Aycock, J., Horspool, R.N.: Faster generalized LR parsing. In: International Conference on Compiler Construction (CC 1999). Volume 1575 of Lecture Notes in Computer Science (LNCS)., Amsterdam, Springer (1999) 32–46
[44] Rozenberg, G., ed.: Handbook of Graph Grammars and Computing by Graph Transformation. Volume 1. World Scientific (1997)
[45] Engelfriet, J., Rozenberg, G.: Node replacement graph grammars. [44] 1–94
[46] Drewes, F., Kreowski, H.J., Habel, A.: Hyperedge replacement graph grammars. [44] 95–162
[47] Nupponen, K.: The design and implementation of a graph rewrite engine for model transformations. Master's thesis, Helsinki University of Technology (2005)
[48] Vainikainen, T.: Applying graph rewriting to model transformations. Master's thesis, Helsinki University of Technology (2005)
[49] Blom, J., Hessel, A., Jonsson, B., Pettersson, P.: Specifying and generating test cases using observer automata. In: Proc. Formal Approaches to Software Testing 2004. Volume 3395 of Lecture Notes in Computer Science. (2005) 125–139
[50] Gotlieb, A., Botella, B., Rueher, M.: Automatic test data generation using constraint solving techniques. In: ISSTA '98: Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis, New York, NY, USA, ACM Press (1998) 53–62
[51] Khurshid, S., Pasareanu, C.S.: Generalized symbolic execution for model checking and testing. In: Tools and Algorithms for the Construction and Analysis of Systems: 9th International Conference, TACAS 2003. Volume 2619 of Lecture Notes in Computer Science. (2003) 553–568
[52] Lee, G., Morris, J., Parker, K., Bundell, G.A., Lam, P.: Using symbolic execution to guide test generation: Research articles. Softw. Test. Verif. Reliab. **15**(1) (2005) 41–61
[53] Dechter, R.: Constraint Processing. Morgan Kaufmann Publishers (2003)
[54] Apt, K.R.: Principles of Constraint Programming. Cambridge University Press (2003)
[55] The Unicode Consortium: The Unicode Standard, Version 5.0. 5th edn. Addison-Wesley Professional (2006)
[56] Jones, R., Lins, R.D.: Garbage Collection: Algorithms for Automatic Dynamic Memory Management. Wiley (1996)
[57] Dechter, R.: Bucket elimination: A unifying framework for reasoning. Artificial Intelligence **113**(1-2) (1999) 41–85
[58] Cunha, J.C., Rana, O.F., eds.: Grid Computing: Software Environments and Tools. 1 edn. Springer (2005)
[59] Cousot, P.: Abstract interpretation. ACM Computing Surveys **28**(2) (1996) 324–328
[60] Bozga, M., Fernandez, J.C., Ghirvu, L.: Using static analysis to improve automatic test generation. In: Tools and Algorithms for Construction and Analysis of Systems. (2000) 235–250