Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 176 (2007) 233–247
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Rewriting Logic Systems

Grit Denker[a] Carolyn Talcott[a] Grigore Rosu[b]
Mark van den Brand[c] Steven Eker[a] Traian Florin Şerbănuţă[b]

[a] *SRI International*
[b] *University of Illinois Urbana-Champaign*
[c] *Technical University Eindhoven (TU/e), Netherlands*

Abstract

We present an overview of rewriting-based systems that were presented at the workshop.

Keywords: Rewriting Competition, Rewriting Systems, ASF+SDF, Maude, TOM + XRHO, MOMENT-OCL, ITP/OCL, ITP

1 Introduction

An important aspect of WRLA is to stimulate discussions of applications of Rewriting Logic and to make the community aware of systems and tools based on Rewriting Logic. Several such systems were demonstrated as part of the workshop. In the following we give an overview of the capabilities of the demonstrated systems. In addition, there was a competition of two mature and widely used algebraic specification environments based on rewriting, namely ASF+SDF and Maude. We present the rules of the competition, describe the benchmarks on which the system were tested and the results of the competition.

2 ASF+SDF

ASF+SDF is a general-purpose, executable, algebraic specification formalism based on (conditional) term rewriting. Its main application areas are the definition of the syntax and the static semantics of (programming) languages, program transformations and analysis, and for defining translations between languages.

The ASF+SDF formalism [11] is a combination of two formalisms: ASF (the Algebraic Specification Formalism) and SDF (the Syntax Definition Formalism). SDF is used to define the concrete syntax of a language, whereas ASF is used to define

doi:10.1016/j.entcs.2007.06.018

conditional rewrite rules; the combination ASF+SDF allows the syntax defined in the SDF part of a specification to be used in the ASF part, thus supporting the use of user-defined syntax when writing ASF equations. ASF+SDF also supports modular structuring of specifications using names modules, and thus enabling reuse.

The ASF+SDF and the ASF+SDF Meta-Environment have been applied in a broad range of applications. The application areas can be characterized as: prototyping of domain specific languages, software renovation, and code generation. An overview of some of the applications is given in [1].

Syntax Definition Formalism

SDF is a declarative formalism used to define the concrete syntax of languages: programming languages, for example Java and Cobol, and specification languages, such as Chi, Elan, and Action Semantics. SDF does not impose any restrictions on the class of grammars used, it accepts arbitrary, lexical, cycle-free, context-free grammars, which may even be ambiguous. Since the class of all context-free grammars is closed under union, a modular definition of grammars is possible in SDF, unlike other (E)BNF formalisms.

Algebraic Specification Formalism

ASF is a declarative formalism used to define the semantics of (programming) languages. It provides conditional equations, also allowing negative conditions. The concrete syntax defined in the corresponding SDF module and in the transitive closure of any imported modules (only the exported sections, of course) can be used when writing the conditional equations of an ASF module. Traversal functions [4] provide a concise way of defining an ASF function which traverse the term and perform program transformation and/or accumulation operations on specific nodes in the underlying term without providing all intermediate rewrite steps. Memo functions for caching repeated computations, list matching and other features are part of ASF.

ASF+SDF **Meta-Environment**

The development of ASF+SDF specifications is supported by an interactive integrated programming environment, the ASF+SDF Meta-Environment [2]. This programming environment provides syntax directed editing facilities for both the SDF and ASF parts of modules as well as for terms, well-formedness checking of modules, interactive debugging of ASF equations, and visualisation facilities of the import graph and parse trees. The ASF+SDF Meta-Environment provides the following features:

- Interactive support for writing a formal specification of a problem.
- An interactive environment for a new (application) language.
- Support for analyzing or transforming programs in existing languages.

3 Maude

Maude is a language and a system based on rewriting logic [9,6,7]. Maude modules are rewrite theories, while computation with such modules corresponds to efficient deduction by rewriting. Since rewriting logic contains equational logic, Maude also supports equational specification and programming in its sublanguage of functional modules and theories. The underlying equational logic of Maude is membership equational logic, that has sorts, subsorts, operator overloading, and partiality definable by membership and equality conditions. Because of its logical basis and its initial model semantics, a Maude module defines a precise mathematical model. This means that Maude and its formal tool environment can be used in three, mutually reinforcing ways: as a declarative programming language, as an executable formal specification language, and as a formal verification system. The Maude system, its documentation, and related papers and applications are available from the Maude website `http://maude.cs.uiuc.edu`.

Maude provides very efficient support for rewriting modulo any combination of associativity, commutativity, and identity axioms, and provides two built-in rewrite strategies: top-down rule fair and position fair. Maude's rewrite engine makes extensive use of advanced semicompilation techniques and sophisticated data structures supporting rewriting modulo. Besides supporting efficient execution, Maude also provides a range of formal tools and algorithms to analyze rewrite theories and verify their properties including a search facility for doing breadth first search with cycle detection, and a linear time temporal logic (LTL) model checker. Note that there is no finite-state assumption: any executable rewrite theory can be analyzed.

Rewriting logic is reflective, in the sense of being able to express its own metalevel at the object level [8]. Reflection is supported efficiently in Maude endowing the language with powerful metaprogramming capabilities, including both user-definable module operations and declarative strategies to guide the deduction process. The Maude `META-LEVEL` provides *descent functions* reifying: the process of reducing a term to normal form; the process of applying a rule to a subject term; the two default strategies for rewriting a term is reified by functions; the process of matching a pattern to a subject term; the process of searching for a particular pattern term; and parsing and pretty printing of a term; and key sort operations. It provides *ascent functions* for obtaining the metarepresentation of terms, and of modules in the module database.

Maude provides a number of advanced features: rewrites in the conditions of a conditional rewrite rules, in addition to match terms and boolean conditions; frozen arguments to control the order of rewriting; on-the-fly declaration of variables, statement attributes (all statements can be given labels for improving tracing, and we can attach an arbitrary string of metadata to a statement for metaprocessing), advanced profiling and debugging features. The module algebra provides operations for summation and renaming of modules, as well as support for parameterized programming by means of theories and views. There are predefined libraries of parameterized data types, supporting efficient versions of lists, sets, maps, and arrays. Maude provides for communication with external objects, a linear Diophantine equation solver; a strategy language to be used at the object level; and support for unification.

Maude includes some built-in functional modules providing convenient high-performance functionality within the Maude system. In particular, the built-in modules of integers, natural, rational and floating-point numbers, quoted identifiers, and strings provide a minimal set of efficient operations for Maude programmers. The built-in natural numbers allow Maude programmers to deal with natural numbers with a C-like performance for simple arithmetic operations on them (using GNU GMP). Built-in natural numbers bridge the gap between clean Peano-like axiomatizations of numbers with an explicit successor function, and rather more efficient binary representations of unbounded natural number arithmetic. Integers are constructed from natural numbers using the unary minus operator. Similarly, the rational numbers are constructed from natural numbers using a division operator. The module of floating-point numbers allows Maude users access to the IEEE-754 double precision floating-point arithmetic when this is supported by the underlying hardware platform. Floats are not algebraic term structures; they are treated as a large set of constants. Maude's built-in strings are based on the SGI rope package which has been optimized for functional programming, where copying with modification is supported efficiently, while arbitrary in-place updates are not. A number of conversion functions is also provided.

4 Rewriting Competition

For the first time at WRLA, and perhaps anywhere, a brute-force speed rewrite engine competition has been organized. We warmly thank Mark van den Brand and Steven Eker for their input and expertize; without their help, the competition would have probably not taken place, or the results would have been biased or irrelevant.

4.1 Why Conducting a Competition?

One can regard rewriting as a generic, natural computational framework, able to seamlessly capture a broad range of other computational models, ranging from foundational ones such as Turing machines and lambda calculi, to complex programming languages and paradigms. Term rewriting gives executable semantics to equational logics at zero-representational distance, but, morover, it is an inherently parallel computational framework, with a high chance to become even more popular in the future, because of the unavoidable technological trends of parallelizing computer architectures, including chips and memory.

The idea of organizing a rewrite competition arose from noticing various applications of rewriting in different areas and by different categories of researchers, many of them manifesting a genuine and explicit interest in term rewriting. We believe that many of us can benefit from such rewrite engine competitions, provided that they are fair and explicitly state what was tested in each case. For example, users of rewrite engine can more informatively select the right rewrite engine for their their particular application. On the other hand, for rewrite engine developers, such events give them ideas on how to improve their tools and what to prioritize, as well as a clearer idea of how their engine compares to others. Finally, term rewriting gets more exposure to colleagues and thus more visibility overall.

Why focus on efficiency? First, for many of us, rewriting is a programming paradigm, our favorite "programming language". Consequently, we expect rewriting to be fast. Second, speed of rewriting is one of the reasons for which algebraic specification and supporting systems gained more attention and popularity today: many find it as a big advantage to design a system formally as a specification and then "execute" it efficiently, reducing to zero the time between design and testing the design. Third, interest for rewrite competitions and speed was already manifested with certain success in other aspects of rewriting - for example termination.

We hope this competition marks the beginning of a long term and *fair* comparison of the performance of various rewrite engines. Specifically, we need to understand, discuss, agree, or disagree upon the relevant *comparison criteria*: raw speed; memory consumption/management; use of parallelism; compiler vs. interpreter; scalability. We believe that the importance one gives to any of these criteria largely depends on the role rewriting plays in the development of their applications. Also we hope these competitions will reveal a *benchmark* of meaningful examples addressing all aspects of rewriting as a programming paradigm.

4.2 The Beginning

To our knowledge, this is the first "official" competition aiming at comparing the raw speed, memory management and built-ins (lists, integers and so on) of rewrite engines. Consequently, our goals were not very ambitious - we just wanted to start the ball rolling, to see whether the community manifests enough interest to continue at this time.

This edition included only two rewrite engines:

- ASF+SDF [14] (represented by Mark van den Brand), and
- Maude [10] (represented by Steven Eker).

We hope to motivate other researchers to contribute to future competitions. To eliminate suspicion and increase fairness of the comparison, the running of the tests was not performed by the rewrite engine developers, but by a graduate student, Traian Florin Şerbănuţă, who is using both ASF+SDF and Maude in his research, each for different purposes. Also, Mark van den Brand and Steven Eker offered to help and revise the specifications and their corresponding formalizations to make sure that the best code was written for each of the experiments.

4.3 Disclaimer

Due to the specificity of the two engines (ASF+SDF is a compiler, Maude is an interpreter), the results should be regarded as a starting point for further discussions rather than being definitive. For example, it is well-known from many other areas that interpreted code tends to be an order of magnitude slower than the corresponding compiled code, even when no aggressive compilation optimizations are employed. Also, we do not claim that we have captured the actual strengths of the systems tested; it was actually quite difficult to devise compelling examples that fit within the intersection of the capabilities of the two systems. For example, ASF+SDF has a *great parser* and was successfully applied in large program transformation projects, which we were not able to test. On the other hand, Maude has *AC matching*, and a *suite of tools* (theorem prover, model checker) making it amenable for formal analysis projects, that we have not tested either.

4.4 Results

For each of the rewrite system examples considered in the competition, we first give the "code" in a generic, mathematical and intuitive notation, and then a table with execution results in the two engines on various inputs. These are followed by some "inside" remarks by Steven Eker, who profiled (using Quantify) Maude's behavior for some of the given tests. These are often completed by Mark van den Brand's remarks as an ASF+SDF insider.

Factorial. This is a simple two-rule rewrite system, calculating the factorial of a natural number:

$$\mathit{fact}(0) \to s(0)$$

$$\mathit{fact}(s(n)) \to s(n) * \mathit{fact}(n)$$

Two versions were considered: one with Peano successor and the other with libraries. With Peano successor, the following numbers have been obtained:

n	8	9	10
ASF+SDF	0.7s	8.28s	>1GB
Maude	12ms	130ms	1.1s

It is surprising that Maude outperforms ASF+SDF since a trivial rewrite system should greatly favor a compiler over an interpreter. Here, Maude runs at close to its top speed (3.5M rewrites/sec).

There are two possible causes for the bad behaviour of ASF+SDF. The first is in the way the terms are represented in the underlying ATerm-library [3]. In case of the successor notion it is very likely that given the term representation the internal hash function of the ATerm-library does not work optimally. A second cause is the restricted machine stack, it is possible to increase the size of the machine stack to solve this problem.

The following numbers have been obtained when the two rewrite engines used their builtin libraries for integers:

n	144	160	20,000
ASF+SDF	1min	2min	>5min
Maude	1ms	1ms	2s

There are no suprises in these results. Maude uses the GNU GMP library as a backend for builtin arithmetic though it maintains the illusion of algebaically defined numbers via a fancy representation and matching and replacement algorithms. Almost all the cpu time goes for bignum multiplications done very efficiently by GMP.

ASF+SDF does not provide built-in integers, it provides a library where the decimal integer arithmetic is specified. So, even with a compiler the performance of Maude is much better.

Bubble-Sort. This is a trivial, but interesting one-rule rewrite system. Many "rewriting programmers" like to show this example to "outsiders", as a sample of how elegant rewriting is as a programming paradigm:

$x, y \rightarrow y, x \Leftarrow y < x$

The above is not only concise, but it can also give a very fast sorting algorithm if run on a parallel rewrite engine. Again, we tested two capabilities of the rewrite engines: with list constructors and with built-in lists. Recall that both ASF+SDF and Maude have specialized rewriting algorithms for associative lists. With list constructors we obtained the following numbers:

n	1,000	2,000	4,000	8,000
ASF+SDF	2.1s	10.8s	50s	210s
Maude	2.6s	10.9s	43s	178s

There is not too much difference between the systems here. Conditional equations are a bit of an "Achilles heel" for Maude. The design of Maude's rewriting engine deliberately trades performance on conditional equations for performance on unconditional equations by preallocating data structures that must be laboriously saved and restored when a condition is encountered.

The following numbers have been obtained when the two engines used their (built-in) associative lists:

n	200	400	800
ASF+SDF	3.9s	35.7s	288s
Maude	5.5s	42.6s	338s

Maude uses multiple special representations for the argument lists of associative operators. However, this comes to naught when conditional equations are involved and associative matching problems have to be solved in the slowest, most general way since potentailly all solutions have to be tried, and thus, they must be generated in a systematic manner. The added generality of a full associative matching algorithm over a list matching algorithm seems to hurt Maude slightly.

List length. There are many rewrite-based programs that traverse lists and do something for each element. We have picked one of the simplest, namely one that counts the number of elements in a list. The rewrite system is trivial, so we do not give it here; we only mention that traversal of the list was from the first element to the last, and that we have used built-in associative lists in both cases:

n	2^7	2^{18}	2^{19}	2^{20}
ASF+SDF	0s	3.7s	8.25s	18.2s
Maude	1ms	3.7s	7.2s	>1GB

Here there are no conditional equations and Maude can usedits persistent associative term representation and fastest associative matching algorithms. However, since the equations only work at the front of the list there is no saving over a more naive list representation and the persistent representation runs out of memory earlier.

Reversing lists. Unlike in standard functional programming where one can only access the head of a list, in rewriting programming with matching modulo assiciativity one can have very elegant definitions of the reverse operation on lists. We have tested the following three one-rule rewrite systems defining list reversal. For both engines we used associative lists.

$$rev_1(X, L) \to rev_1(L), X$$

$$rev_2(L, X) \to X, rev_2(L)$$

$$rev_3(X, L, Y) \to Y, rev_3(L), X$$

The following numbers have been obtained for rev_1:

n	2^{11}	2^{12}	2^{13}
ASF+SDF	11s	62s	>5min
Maude	10ms	20ms	50ms

The following numbers have been obtained for rev_2:

n	2^{11}	2^{12}	2^{13}
ASF+SDF	10s	53s	275s
Maude	10ms	20ms	40ms

The following numbers have been obtained for rev_3:

n	2^{11}	2^{12}	2^{13}
ASF+SDF	11s	55s	264s
Maude	10ms	20ms	40ms

Here there are no conditional equations and the equations work at both ends of the list. This is where the persistent associative term representation and its constant time matching and replacement algorithms really shine. Consequently Maude runs all 3 rewrite systems in the linear time.

In ASF+SDF lists are directly mapped to the list representation of the ATerm library [3]. Operations on the head of these lists are extremely efficient, O(1), whereas operations, such as adding new elements, at the tail of the list are extremely expensive, $O(n^2)$ where n is the length of the list. This explains the difference in behavior between the two systems.

Quick-sort. Quick-sort exercises matching in conditions, a capability that both ASF+SDF and Maude have. We used constructor-based lists (not associative) and a standard and easy to define append:

$$split(x,(y,L)) \rightarrow \langle LTx,(y,GTx)\rangle \Leftarrow x < y \wedge split(x,L) \rightarrow \langle LTx,GTx\rangle$$

$$split(x,(y,L)) \rightarrow \langle (y,LTx),GTx\rangle \Leftarrow x \geq y \wedge split(x,L) \rightarrow \langle LTx,GTx\rangle$$

$$split(x,nil) \rightarrow \langle nil,nil\rangle$$

$$qsort(nil) \rightarrow nil$$

$$qsort(x,L) \rightarrow append(qsort(LTx),(x,qsort(GTx))) \Leftarrow split(x,L) \rightarrow \langle LTx,GTx\rangle$$

The following numbers have been obtained:

n	5,000	10,000	15,000
ASF+SDF	0.8s	4.5s	14.6s
Maude	24s	101s	224s

Brute force rewriting with conditions is Maude's weakest suit. For brute force rewriting it is hard for an interpreter to compete with a compiler. Furthermore, Maude has to carry runtime type information around even though it is never used. As explained above, condition evaluation is very expensive.

Odd/Even. This is an artificial example testing the exponential explosion that can result because of conditional rewriting:

$$odd(0) \rightarrow false$$

$$even(0) \rightarrow true$$

$$odd(s(i)) \rightarrow true \Leftarrow even(i) == true$$

$$even(s(i)) \rightarrow true \Leftarrow odd(i) == true$$

$$odd(s(i)) \rightarrow false \Leftarrow even(i) == false$$

$$even(s(i)) \rightarrow false \Leftarrow odd(i) == false$$

n	23	24	25	26	27
ASF+SDF	0s	0s	0s	0s	0s
Maude	0s	77s	0s	>5min	0s

Maude attempts the conditions in the order given so evaluating odd() on an odd number results in an almost immediate success while evalutating odd() on a even number results in an exponential search. Combining the evaluations of odd() and even() would require inter-equation analysis which currently is not performed in Maude. Alternatively, adding the memo attribute to odd() and even() avoids the exponential blow-up and allows the even cases to run in 0s.

ASF+SDF did quite well in this example because it optimizes the compiled code by avoiding re-computation of the same rewrite sequence in conditions.

ASF+SDF *TOPLAS benchmark.* This benchmark was used in [14] to study resource usage for brute-force (that is, no built-ins, no strategies) rewriting on rewrite engines and on programming languages implemented using rewriting. Below is the benchmarks' description:

benchsym performs symbolic evaluation of 2^n modulo 17. It aims on testing speed of rewriting with almost no memory usage.

bencheval also performs symbolic evaluation of 2^n modulo 17, but first it expands the expression without evaluating it. This is done to test memory management.

benchtree computing on huge (2^n), not-alike trees. Also done to test memory management.

The following numbers have been obtained for *benchsym*:

n	25	26	27
ASF+SDF	17.5s	32.8s	65.6s
Maude	132.8s	263.6s	>5min

Maude runs at 5.3 million rewrites/second which is about as fast as it ever runs. It is about 8 times slower than ASF+SDF which illustrates the huge advantage of compilation for eager, free theory rewriting. In compiled code, the lefthand sides of the equations for each symbol can be compiled into a nesting of switch statements while each right hand side becomes nesting of function calls, with actual terms only being constructed when they are fully reduced. In constrast, Maude explicitly does each replacement and keeps a flag to hold the reduction status of each subterm. The user can interrupt Maude at any moment with control-C and examine the current term.

The following numbers have been obtained for *bencheval*:

n	21	22	23
ASF+SDF	1.5s	3s	7.8s
Maude	14.7s	29.1s	>1GB

The heuristic Maude uses to decide when to allocate fresh memory is biased towards allocating memory that is not strictly needed in order to decrease the frequency of garbage collection. In these examples this approach appears to hurt performance (probably because of L2 cache effects), making it about 10 times slower than ASF+SDF and ultimately causes it to run out of memory.

The following numbers have been obtained for *benchtree*:

n	19	20	21
ASF+SDF	0.3s	0.6s	1.1s
Maude	8.3s	16.4s	>1GB

In addition to the problem noted for bencheval, Maude spends a large part of the cpu time doing setup and tear down of contexts for evaluating conditions, making it about 28 times slower than ASF+SDF.

ASF+SDF outperforms Maude mainly because of the internal representation of the terms. The maximal subterm sharing of the ATerm-libary ensures a minimal use of memory at least in the bencheval and benchtree.

5 System Demonstrations

5.1 ITP/OCL and Web ITP Tool Server

5.1.1 ITP/OCL: A Rewriting-Based Validation Tool for UML+OCL Class Diagrams

System demonstration presented by Manuel Clavel (Universidad Complutense de Madrid, Spain)

This presentation introduced the ITP/OCL tool, a rewriting-based tool that supports automatic validation of static class diagrams with respect to OCL constraints. The ITP/OCL tool is directly based on the equational specification of UML+OCL class diagrams. It is written entirely in Maude making extensive use of its reflective capabilities. The Visual ITP/OCL was also presented. This is a Java graphical interface that can be used as a front-end for the ITP/OCL tool.

5.1.2 Web ITP Tool Server: A Web-Based Interface for the ITP Tool

System demonstration presented by Adrian Riesco (Universidad Complutense de Madrid, Spain).

The ITP tool is an experimental inductive theorem prover for proving properties of (Diet) Maude equational specifications, i.e., specifications in membership equational logic. The Web ITP tool is a client-server application that allows a web-based interaction with the ITP tool. It includes a (Diet) Maude module editor, an ITP formula editor, and an ITP command editor. The Web ITP tool has been developed as an educational project: it aims to provide an ITP-based tool for teaching specification, validation and verification of abstract data types.

5.2 New Features of Maude 2.2

System demonstration presented by Steven Eker (SRI International, USA).

New features introduced with the recent 2.2 release of Maude were demonstrated. These included Core Maude support for parameterized modules, predefined container data types, random number generation, counters, external objects, and the built-in linear Diophantine equation solver. Features planned for Maude 2.3 were also previewed.

For random number generation we use the Mersenne Twister as a source of high quality random numbers. The sequence of numbers generated from a particular seed is viewed as a function *random* from the natural numbers into $[0, \ldots, 2^{32} - 1]$. Internally, the state of the Mersenne Twister is cached so that if *random* was last called on n, calling it on $n + k$ requires k steps of the twister. Usually $k = 1$. The seed is set at startup time with the command line flag -random-seed=$\langle int \rangle$ so the sequence is constant for any given run of Maude.

It is often useful to have implicitly stored state, especially when working with random numbers. Such state cannot be functional but is available in system modules via counters. *counter* is a special constant of kind [Nat] that each time it is rewritten (by rules) generates the next larger natural number. It can be viewed as a special built-in strategy for executing the otherwise nonexecutable rule:

```
rl counter => N:Nat [nonexec] .
```

Additional counters can be created using renamed copies.

Maude 2.2 supports external objects to represent entities in the external world. Configurations that want to communicate with external objects must contain at least one portal, an object of the predefined sort Portal.

The command *erewrite* is used to start rewrite sequences that may involve external objects. It may not terminate even if no rewrites are possible. This is because there may be incomplete transactions with external objects and more rewriting may be possible once they complete.

Maude 2.2 supports IPv4 TCP client and server sockets. Sockets are created by sending a message to a special external object called *socketManager*. *send* and *receive* messages can then be sent to the newly created socket objects. Message are usually paired: a user object sends a message to an external object and waits for a reply message. This enforces sequentialization in the otherwise concurrent Maude configuation.

In the linear Diophantine solver, integer vectors and matrices are implemented as instantiations of the ARRAY module. The predefined module DIOPHANTINE contains a solver for non-negative solutions of (homogenous and inhomogenous) linear Diophantince equations. The solution is a pair of sets of integer vectors A and B. The non-negative solutions are formed by adding a vector from A to a non-negative linear combination of vectors from B. Two algorithms are currently implemented: Contejean-Devie and a method based on Gaussian elimination and extended gcd.

A built-in strategy language is planned for Maude 2.3. Rewriting a term with a strategy can be invoked using a strategy expression. Several leaf strategies have already been implemented: apply a rule with a specific label, apply any rule, apply a rule with a given label and a given substitution, produce identity rewrite or the empty set of successors. These strategies can be combined with operators such as sequence, repeat strategy 0/1 or more times, and choice.

5.3 MOMENT-OCL: Algebraic Specifications of OCL 2.0 within the Eclipse Modeling Framework

System demonstration presented by Artur Boronat, Joaquín Oriente, Abel Gómez, José Á. Carsí, and Isidro Ramos (U. Politècnica de Valencia, Spain).

Model-Driven Development is a field in Software Engineering that, for several years, has been representing software artifacts as models in order to improve productivity, quality, and economy. Models provide a more abstract description of a software artifact than the final code of the application. Interest in this field has grown in software development companies such as the Model-Driven Architecture (MDA), supported by OMG, and the Software Factories, supported by Microsoft, ensuring a model-driven technology stock for the near future.

Model-Driven Development has evolved to the Model-Driven Engineering field, where not only design and code generation tasks are involved, but also traceability, model management, meta-modeling issues, model interchange and persistence, etc. To fulfill these requirements, model transformations and model queries are relevant issues that must be addressed. In the MDA context, they are handled from an open-standard point of view. The standard Meta-Object Facilities (MOF) provides a way to define meta-models. The standard proposal Query/Views/Transformations (QVT) indicates how to provide support for both transformations and queries. In QVT, while new languages are provided for model transformation, the Object Constraint Language (OCL) remains the best choice for queries.

OCL is a textual language that is defined as a standard "add-on" to the UML standard. It is used to define constraints and queries on UML models, allowing the definition of more precise and more useful models. It can also be used to provide support for meta-modeling (MOF-based and Domain Specific Meta-modeling), model transformation, Aspect-Oriented Modeling, support for model testing and simulation, ontology development and validation for the Semantic Web, among others.

Despite its many advantages, while there is wide acceptance for UML design in CASE tools, OCL lacks a well-suited technological support.

In this demonstration, we present the MOMENT-OCL tool, which integrates an algebraic specification of the operational semantics of part of the OCL 2.0 standard into the Eclipse Modeling Framework (EMF). EMF is a modeling environment that is plugged into the Eclipse platform and that provides a sort of implementation of the MOF. EMF enables the automatic import of software artifacts from heterogeneous data sources: UML models, relational schemas, and XML schemas. In MOMENT-OCL, OCL queries and invariants can be executed over instances of EMF models in Maude. An interesting feature of this algebraic specification of the OCL 2.0 is the use of the parameterization to reuse the OCL specification for any meta-model/model and the simulation of higher-order functions for the sake of the reuse of collection operator definitions.

5.4 TOM + XRHO the explicit rewriting calculus.

System demonstration presented by Germain Faure and Antoine Reilles (LORIA, FR).

Following the experience of Elan [12], the Tom [13] language was developed to provide rewrite tools for implementation of calculi, for compilation and for XML-transformations. The demonstration focussed on the former. Tom provides a language to define a syntax (a signature) embedded into Java. One can perform pattern matching with support of associative matching modulo neutral element (also known as list-matching). In addition, we can guide the application of rules with a strategy language defining term traversals (namely evaluation/rewriting strategies).

The originality of Tom lies in the combination of formal aspects with a general purpose language (such as Java). This combination leads to an agile language. At the same time, the strategy language inspired by Elan and Stratego [15] gives the opportunity to reduce the code written in the general purpose language (and thus increases the formal parts).

The goal of TOM is to facilitate making trustable and modular implementations of calculi rewriting systems. TOM was illustrated with an implementation of the explicit rewriting calculus, introduced at WRLA 2004 [5]. This example demonstrated the adequacy of Tom for such a development, enhanced by the integration in a general purpose language and the strategy language.

References

[1] M.G.J. van den Brand. Applications of the ASF+SDF meta-environment. In R. Lämmel, J. Saraiva, and J. Visser, editors, *Generative and Transformational Techniques in Software Engineering*, volume 4143 of *LNCS*, pages 278–296. Springer-Verlag, 2006.

[2] M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a component-based language development environment. In R. Wilhelm, editor, *CC'01*, volume 2027 of *LNCS*, pages 365–370. Springer-Verlag, 2001.

[3] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P. Olivier. Efficient Annotated Terms. *Software, Practice & Experience*, 30:259–291, 2000.

[4] M.G.J. van den Brand, P. Klint, and J.J. Vinju. Term rewriting with traversal functions. *ACM Transactions on Software Engineering and Methodology*, 12(2):152–190, 2003.

[5] Horatiu Cirstea, Germain Faure, and Claude Kirchner. A rho-calculus of explicit constraint application. In *Proceedings of the 5th workshop on rewriting logic and applications.* vol. 117 of Electronic Notes in Theoretical Computer Science, 2004.

[6] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 system. In R. Nieuwenhuis, editor, *Rewriting Techniques and Applications, 14th Intl. Conference*, volume 2706 of *LNCS*, pages 76–87. Springer, 2003.

[7] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude, A High-Performance Logical Framework*, volume 4350 of *LNCS*. Springer, 2007.

[8] M. Clavel and J. Meseguer. Reflection in conditional rewriting logic. *Theoretical Computer Science*, 285(2):245–288, 2002.

[9] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.

[10] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott. The maude 2.0 system. In Robert Nieuwenhuis, editor, *RTA*, volume 2706 of *Lecture Notes in Computer Science*, pages 76–87. Springer, 2003.

[11] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing.* World Scientific, 1996.

[12] Hélène Kirchner and Pierre-Etienne Moreau. Promoting rewriting to a programming language: A compiler for non-deterministic rewrite programs in associative-commutative theories. *Journal of Functional Programming*, 11(2):207–251, 2001.

[13] Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. A Pattern Matching Compiler for Multiple Target Languages. In G. Hedin, editor, *12th Conference on Compiler Construction, Warsaw (Poland)*, volume 2622 of *LNCS*, pages 61–76. Springer-Verlag, MAY 2003.

[14] Mark van den Brand, Jan Heering, Paul Klint, and Pieter A. Olivier. Compiling language definitions: the asf+sdf compiler. *ACM Trans. Program. Lang. Syst.*, 24(4):334–368, 2002.

[15] E. Visser. Stratego: A language for program transformation based on rewriting strategies (system description). In A. Middeldorp, editor, *Rewriting Techniques and Applications, 12th International Conference*, LNCS 2051, pages 357–362, Utrecht, The Netherlands, May 22–24, 2001. Springer.