Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science 124 (2005) 17–28

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Invariant-Driven Strategies for Maude[1]

Francisco Durán[2] Manuel Roldán[3] Antonio Vallecillo[4]

Dpto. de Lenguajes y Ciencias de la Computación
Universidad de Málaga
Málaga, Spain

Abstract

We propose generic invariant-driven strategies that control the execution of systems by guaranteeing that the given invariants are satisfied. Our strategies are generic in the sense that they are parameterized by the system whose execution they control, by the logic in which the invariants are expressed, and by the invariants themselves. We illustrate the use of the strategies in the case of invariants expressed in propositional logic. However, the good properties of Maude as a logical and semantic framework, in which many different logics and formalisms can be expressed and executed allow us to use other logics as parameter of our strategies.

Keywords: Execution strategies, Maude, rewriting logic, reflection.

1 Introduction

To deal with nonterminating and nonconfluent systems, we need good ways of controlling the rewriting inference process. In this line, different languages offer different mechanisms, including approaches based on metaprogramming, like Maude [2,3], or on strategy languages, like ELAN [1]. However, although the separation of logic and control greatly simplifies such a task, these mechanisms are sometimes hard to use, specially for beginners, and usually compromise fundamental properties like extensibility, reusability, and maintainability.

[1] Partially supported by projects TIC 2001-2705-C03-02 and 2002-04309-C02-02
[2] Email: duran@lcc.uma.es
[3] Email: mrc@lcc.uma.es
[4] Email: av@lcc.uma.es

doi:10.1016/j.entcs.2004.11.018

Formalisms like Z and UML suggest an interesting alternative, since they allow to define invariants or constraints as part of the system specifications. Although executing or simulating Z specifications may be hard, we can still find tools like Possum [9] or Jaza [12], which can do a reasonable simulation of such specifications. We find something somehow similar in UML, where, by specifying OCL constraints on our specifications, they can be made executable [13].

The execution or simulation of specifications with constraining invariants is typically based on integrating somehow the invariants into the system code. However, such an integration is clearly unsatisfactory: the invariants get lost amidst the code, and become difficult to locate, trace, and maintain. Moreover, the programs and the invariants to be satisfied on them are usually expressed in different formalisms, and *live at different levels of abstraction*: invariants are defined *on* programs. Therefore, it is interesting to have some way of expressing them separately, thus avoiding the mixing of invariants and code.

Maude does not provide direct support for expressing execution invariants. However, it does provide reflective capabilities and support to control the execution process, being also an excellent tool in which to create executable environments for various logics and models of computation [4]. Thus, it turns out to be a very good candidate for giving support to different types of invariants, which may be expressed in different formalisms.

In this paper we propose generic invariant-driven strategies to control the execution of systems by guaranteeing that the given invariants are always satisfied. Our strategies are generic in the sense that they are parameterized by the system whose execution they control, by the logic in which the invariants are expressed, and by the invariants themselves. The good properties of Maude as a logical and semantic framework [8], in which many different logics and formalisms can be expressed and executed, allow us to say that other logics and formalisms may be used as parameters of our strategies. We will use in this paper the case of propositional logic, although we have also experimented with future time linear temporal logic.

The paper is structured as follows. Section 2 serves as a brief introduction to rewriting logic and Maude. Section 3 introduces the definition of strategies in Maude, and serves as a basis for the introduction of invariant-guided strategies in Section 4. Section 5 describes as an example the case of invariants expressed using propositional calculus. Finally, Section 6 draws some conclusions.

2 Rewriting Logic and Maude

Maude [2,3] is a high-level language and a high-performance interpreter and compiler in the OBJ [5] algebraic specification family that supports membership equational logic [11] and rewriting logic [10] specification and programming of systems.

Membership equational logic is a Horn logic whose atomic sentences are equalities $t = t'$ and *membership assertions* of the form $t : S$, stating that a term t has sort S. Such a logic extends order-sorted equational logic, and supports sorts, subsort relations, subsort polymorphic overloading of operators, and the definition of partial functions with equationally defined domains.

Rewriting logic is a logic of change that can naturally deal with state and with highly nondeterministic concurrent computations. In rewriting logic, the state space of a distributed system is specified as an algebraic data type in terms of an equational specification (Σ, E), where Σ is a signature of sorts (types) and operations, and E is a set of (conditional) equational axioms. The dynamics of a system in rewriting logic is then specified by rewrite *rules* of the form $t \to t'$, where t and t' are Σ-terms. These rules describe the local, concurrent transitions possible in the system, i.e. when a part of the system state fits the pattern t then it can change to a new local state fitting pattern t'. Rules may be conditional, in which case the guards act as blocking pre-conditions, in the sense that a conditional rule can only be fired if the condition is satisfied.

In Maude, object-oriented systems are specified by object-oriented modules in which classes and subclasses are declared. A class is declared with the syntax

$$\texttt{class}\ C\ \texttt{|}\ a_1 : S_1\texttt{,}\ \ldots\texttt{,}\ a_n : S_n,$$

where C is the name of the class, a_i are attribute identifiers, and S_i are the sorts of the corresponding attributes. Objects of a class C are then record-like structures of the form

$$\texttt{<}\ O\ :\ C\ \texttt{|}\ a_1 : v_1\texttt{,}\ \ldots\texttt{,}\ a_n : v_n\ \texttt{>},$$

where O is the name of the object, and v_i are the current values of its attributes. Objects can interact in a number of different ways, including message passing. Messages are declared in Maude in `msg` clauses, in which the syntax and arguments of the messages are defined.

In an object-oriented system, a state, which is called a *configuration*, has the structure of a multiset made up of objects and messages that evolves by rewriting using rules that describe the effects of the communication events of objects and messages. The general form of such rewrite rules is

```
crl [r] :
  < O1 :  C1 | atts1 > ... < On :  Cn | attsn >
  M1 ... Mm
  => < Oi1 :  C'i1 | atts'i1 > ... < Oik :  C'ik | atts'ik >
     < Q1 :  C''1 | atts''1 > ... < Qp :  C''p | atts''p >
     M'1 ... M'q
  if Cond .
```

where r is the rule label, $M_1...M_m$ and $M'_1...M'_q$ are messages, $O_1...O_n$ and $Q_1...Q_p$ are object identifiers, $C_1...C_n$, $C'_{i_1}...C'_{i_k}$ and $C''_1...C''_p$ are classes, $i_1...i_k$ is a subset of $1...n$, and $Cond$ is a Boolean condition (the rule's *guard*). The result of applying such a rule is that: (a) messages $M_1...M_m$ disappear, i.e., they are consumed; (b) the state, and possibly the classes of objects $O_{i_1}...O_{i_k}$ may change; (c) all the other objects O_j vanish; (d) new objects $Q_1...Q_p$ are created; and (e) new messages $M'_1...M'_q$ are created, i.e., they are sent. Rule labels and guards are optional.

For instance, the Maude module `DINING-PHILOSOPHERS` below specifies the well known problem of the hungry philosophers. The problem assumes five philosophers sitting around a table, on which five plates and five chopsticks are laid out. A philosopher can do two things, either think, or eat. When he thinks, a philosopher does not need the chopsticks; on the other hand, when thinking, he ends up being hungry. To eat, he needs the two chopsticks which are disposed on each side of his plate. Once he has finished eating, the philosopher releases the chopsticks and starts thinking, and then will be hungry again, etc.

Philosophers are modeled using a class with two attributes. The attribute `state` represents the state of the philosopher—which can be `thinking`, `hungry`, or `eating`—and the attribute `sticks` represents the number of chopsticks he holds. Moreover, a message `chopstick(`N`)` has been defined, indicating that the chopstick N is free. Philosophers and chopsticks are named with numbers from one to five, in such a way that the chopsticks besides the philosopher i are i and $i+1$, or i and 1 if i is 5. Note the subsort declaration `Nat < Oid` making a natural number a valid object identifier.

The system behavior is defined by four rules, each one representing a local transition of the system. For example, the rule labeled as `grab` may be fired when a philosopher object I is hungry and it receives a message indicating that the chopstick J is free, being the chopstick J one of the chopsticks I can grab. As a result, the message is consumed, and the number of chopsticks grabbed by the philosopher is increased. The syntax for rules and conditional rules is, respectively, `rl [`l`] :` t `=>` t' and `crl [`l`] :` t `=>` t' `if` c, with l a rule label, t and t' terms, and c a rule condition.

```
(omod DINING-PHILOSOPHERS is
    protecting NAT .
    subsort Nat < Oid . *** Natural numbers valid object identifiers

    sort Status .
    ops thinking hungry eating : -> Status [ctor] .

    class Philosopher | state : Status, sticks : Nat .
    msg chopstick : Nat -> Msg .

    vars I J K : Nat .

    op _can‘use_ : Nat Nat -> Bool .
    eq I can use J
      = (I == J) or (s(I) == J) or (I == 5 and J == 1) .

    rl [hungry] :
      < I : Philosopher | state : thinking >
      => < I : Philosopher | state : hungry > .
    crl [grab] :
      < I : Philosopher | state : hungry, sticks : K >
      chopstick(J)
      => < I : Philosopher | sticks : K + 1 >
      if I can use J .
    rl [eat] :
      < I : Philosopher | state : hungry, sticks : 2 >
      => < I : Philosopher | state : eating > .
    rl [full] :
      < I : Philosopher | state : eating >
      => < I : Philosopher | state : thinking, sticks : 0 >
         chopstick(I)
         chopstick(s(I)) .
  endom)
```

In Maude, those attributes of an object that are not relevant for an axiom do not need to be mentioned. Attributes not appearing in the right-hand side of a rule will maintain their previous values unmodified.

3 Execution strategies in Maude

System modules and object-oriented modules in Maude do not need to be Church-Rosser and terminating, therefore the system state may evolve in different directions depending on the order in which we apply the rules describing such a system. Maude provides two built-in strategies: The *rewrite* command follows a top-down lazy rule-fair strategy, and the *frewrite* command follows a position-fair bottom-up strategy. Although enough in many cases, the rewriting inference process could not terminate or go in many undesired directions. Thanks to the reflective capabilities that Maude provides, we can define our own strategies, which in fact are defined using statements in a normal module.

Maude provides key metalevel functionality for metaprogramming and for writing execution strategies. In general, strategies are defined in extensions of the predefined module `META-LEVEL` by using predefined functions in it, like `metaReduce`, `metaApply`, `metaXapply`, etc. as building blocks. `META-LEVEL` also provides sorts `Term` and `Module`, so that the representations of a term T

and of a module M are, respectively, a term $\overline{T}$ of sort `Term` and a term $\overline{M}$ of sort `Module`. Constants (resp. variables) are metarepresented as quoted identifiers that contain the name of the constant (resp. variable) and its type separated by a dot (resp. colon), e.g., `'true.Bool` (resp. `'B:Bool`). Then, a term is constructed in the usual way, by applying an operator symbol to a comma-separated list of terms. For example, the term `S |= True` of sort `Bool` in the module `PL-SATISFACTION` below is metarepresented as the term `'_|=_['S:State, 'True.Formula]` of sort `Term`.

Of particular interest for our current purposes are the partial functions `metaReduce` and `metaXapply`.[5]

```
op metaReduce : Module Term ~> Term .
op metaXapply : Module Term Qid ~> Term .
```

`metaReduce` takes a module $\overline{M}$ and a term $\overline{T}$, and returns the metarepresentation of the normal form of T in M, that is, the result of reducing T as much as possible using the equations in M. `metaXapply` takes as arguments a module $\overline{M}$, a term $\overline{T}$, and a rule label L, and returns the metarepresentation of the term resulting from applying the rule with label L in M on the term T.

To illustrate the general approach, and as a first step towards our final goal, let us suppose that we are interested in a strategy that rewrites a given term by applying on it all the rules in a given module, in any order. The strategy should just try to apply the rules one by one on the current term until it gets rewritten. Once a rule can be applied on it, the term resulting from such an application becomes the current term, and we start again. If none of the rules can be applied on a term, then it is returned as the result of the rewriting process. Such a strategy can be specified as follows:

```
op rew : Module Term -> Term .
op rewAux : Module Term ContStruct -> Term .

eq rew(M, T)
  = rewAux(M, T, cont(M)) .

ceq rewAux(M, T, C)
  = T
  if final(C) .
ceq rewAux(M, T, C)
  = if T' :: Term
    then rewAux(M, T', reset(C))
    else rewAux(M, T, C')
    fi
  if C' := next(C)
     /\ T' := metaXapply(M, T, getLabel(C')) .
```

The operation `rew` takes two arguments: the (metarepresentation of) the module describing the system whose execution we wish to control, and the

[5] We have simplified the form of these functions for presentation purposes, since we do not need here their complete functionality. See [3] for the actual descriptions.

term representing the initial state of the system. `rewAux` takes three arguments: the module describing the system, the term being rewritten, and a continuation structure with the labels of the rules in the module, which allows us to iterate on the labels in some order. The strategy gives as result a term which cannot be further rewritten.

In the equations defining `rew` and `rewAux` we assume a function `cont` that takes a module and returns a continuation structure for it, a structure which contains the module's rule labels and keeps control on the last label requested. We also assume the following functions on the sort `ContStruct` of continuation structures: `final`, which returns a Boolean value indicating whether there are more labels in the structure; `reset`, which initializes the structure, that is, it returns the structure with the next label set to be the first one; `next`, which returns the structure with the next label set to be the next one; and `getLabel`, which returns the next label in the sequence. Note that we do not assume a concrete structure; depending on the particular structure used, and on the definition of these operations, the order in which the labels are considered may be different, which provides extra adaptability for our strategy.

Note the use of the `metaXapply` function. A rewriting step $T \xrightarrow{L} T'$ is accomplished only if the rule labeled L is applicable on the term T, being T' the term returned by `metaXapply(`M`, `T`, `L`)`. The membership assertion "T' `::  Term`" is used to check whether the result of the application of the rule is of sort `Term` or not. Note that in case the rule cannot be applied, `metaXapply` returns an error term in a supersort of `Term`.

4 Using invariants to guide the system execution

Basically, an invariant is a property that a specification or program always requires to be true. Instead of using an external monitor to verify a given system specification against an invariant, we propose using invariants as part of our specifications, making it *internal*. We suggest exploiting the possibility of defining execution strategies to *drive* the system execution in such a way that we can guarantee that every obtained state complies with the invariant, thus avoiding the execution of actions conducting the system to states not satisfying the invariant. If we want to define a strategy which guarantees the invariant, we may use a variant of the strategy in Section 3: We just need to check that the invariant is satisfied by the initial state and by every candidate to new state in a rewriting step.

To implement this new strategy we assume a satisfaction Boolean predicate `_|=_` such that, given a state of the system S and an invariant I, then S `|=` I evaluates to `true` or `false` depending on whether the state S satisfies the invariant I or not. The new strategy requires two additional parameters, namely (the metarepresentation of) the invariant predicate, and (the metarepresentation of) the module defining the satisfaction relation in the logic used for expressing such an invariant:

```
op rewInv : Module Module Term Term ~> Term .
op rewInvAux : Module Module Term Term ContStruct -> Term .

ceq rewInv(M, M', T, I)
  = rewInvAux(M, M', T, I, cont(M))
  if metaReduce(M', '_|=_[T, I]) = 'true.Bool .

ceq rewInvAux(M, M', T, I, C)
  = T
  if final(C) .
ceq rewInvAux(M, M', T, I, C)
  = if T' :: Term
       and-then metaReduce(M', '_|=_[T', I]) == 'true.Bool
    then rewInvAux(M, M', T', I, reset(C))
    else rewInvAux(M, M', T, I, next(C))
    fi
  if L := getLabel(next(C))
     /\ T' := metaXapply(M, T, L) .
```

Now the auxiliary function is invoked if the initial state satisfies the invariant. Notice that the operator `rewInv` is declared using `~>`, meaning that if not reduced, it will return an error term of sort `[Term]`, which represents the kind of the sort `Term` and all sorts in its connected component.[6] A kind is semantically interpreted as the set containing all the well-formed expressions in the sorts determining it, and also error expressions. Moreover, the strategy takes a rewriting step only if the term can be rewritten using a particular rule and it yields to a next state which satisfies the invariant. An invariant I is checked by evaluating the expression T' `|=` I, for a given candidate transition $T \xrightarrow{L} T'$. Note however that the rewriting process takes place at the metalevel, and we use `metaReduce` for evaluating the satisfaction of the property.

Notice also that the rules describing the system can be written independently from the invariants applied on them, and the module specifying the system is independent of the logic in which the invariants are expressed, thus providing the right kind of independence and modularity between the system definition and the system invariants. In fact, the strategy is parameterized by the system to be executed (M), the invariant to be preserved (I), and the module defining the satisfaction relation (M'). This allows using different logics to express the invariant without affecting the strategy or the system to execute.

[6] Note that the operator `rew` in Section 3 was declared using `->`, on sorts, since it always returns a term. In the case of `rew` the original term is a valid state, and therefore is can always be given as result.

5 Defining logics for driving the system execution: the propositional calculus

The logic in which the invariants are expressed is independent of the system to be executed. This would allow us to use one logic or another to express our invariants depending on our needs. We illustrate our approach with propositional logic.

If we want to use a specific logic to express the invariant predicates, we need to define the syntax of such a logic and a satisfaction relation for it. Given a set of atomic propositions, which corresponds to the sort `Proposition`, the following module `PROPOSITIONAL-CALCULUS` defines the formulae of the propositional calculus.

```
(fmod PROPOSITIONAL-CALCULUS is
   sort Proposition Formula .
   subsort Proposition < Formula .

   ops True False : -> Formula .
   op _and_ : Formula Formula -> Formula [assoc comm prec 55] .
   op _or_ : Formula Formula -> Formula [assoc comm prec 59] .
   op _xor_ : Formula Formula -> Formula [assoc comm prec 57] .
   op not_ : Formula -> Formula [prec 53] .
   op _implies_ : Formula Formula -> Formula [prec 61 gather(e E)] .
   op _iff_ : Formula Formula -> Formula [assoc prec 63] .

   vars A B C : Formula .

   eq True and A = A .
   eq False and A = False .
   eq A and A = A .
   eq False xor A = A .
   eq A xor A = False .
   eq A and (B xor C) = A and B xor A and C .
   eq not A = A xor True .
   eq A or B = A and B xor A xor B .
   eq A implies B = not(A xor A and B) .
   eq A iff B = A xor B xor True .
 endfm)
```

The module `PROPOSITIONAL-CALCULUS` introduces the sort `Formula` of well-formed propositional formulae, with two designated formulae, namely `True` and `False`, with the obvious meaning. The sort `Proposition`, corresponding to the set of atomic propositions, is declared as subsort of `Formula`. `Proposition` is by the moment left unspecified; we shall see below how such atomic propositions are defined for a given system module. Then, the usual operators are declared. These declarations follow quite closely the definition of Boolean values in Maude and OBJ3 [5], which are based on the decision procedure proposed by Hsiang [7]. This procedure reduces valid propositional formulae to the constant `True`, and all the others to some canonical form which consists of an exclusive or of conjunctions.

The following module `PL-SATISFACTION` defines a satisfaction relation for propositional formulae.

```
(fmod PL-SATISFACTION is
   protecting PROPOSITIONAL-CALCULUS
   sorts State Formula .
   op _|=_ : State Formula -> Bool .

   var  S : State .
   vars F F' : Formula .

   eq S |= (F and F') = (S |= F) and (S |= F') .
   eq S |= (F xor F') = (S |= F) xor (S |= F') .
   eq S |= not F = not S |= F .
   eq S |= True = true .
   eq S |= False = false .
 endfm)
```

As said above, the satisfaction relation `_|=_` is a Boolean predicate such that, given a state (the sort `State` will be defined for each particular problem) and a formula, evaluates to `true` or `false` depending on whether the given state satisfies such a formula or not. Notice that `_|=_` takes a propositional formula as second argument and returns a Boolean value, being `Bool` a predefined sort in Maude.

If we want to use propositional calculus to define invariant predicates for a given problem, we need to define the atomic propositions of interest for such a problem. For example, we could define an invariant predicate for guiding the execution of our philosophers example in such a way that we avoid deadlock situations. The system would go into deadlock if we reach a state where each philosopher has one chopstick. We define what a `State` is—in this example, a `Configuration`—and the proposition `fork(`P`, `N`)`, which holds if the philosopher P has N chopsticks, in the following module:

```
(omod DINING-PHILOSOPHERS-PL-PREDS is
   protecting DINING-PHILOSOPHERS .
   including PL-SATISFACTION .
   subsort Configuration < State .
   op forks : Oid Nat -> Proposition .

   vars I N M : Nat .
   var  C : Configuration .

   eq < I : Philosopher | sticks : N > C |= forks(I, M)
     = N == M .
 endom)
```

Once we have defined the atomic proposition `forks`, it may be used to define the intended invariant for guiding the execution. Thus, the invariant to avoid deadlock states may be expressed as follows:

```
 deadlock-free = not(forks(1, 1)
                     and forks(2, 1)
                     and forks(3, 1)
                     and forks(4, 1)
                     and forks(5, 1))
```

Let us denote $\overline{t}$ and $\overline{M}$ the metarepresentations of a term t and a module

M. We can rewrite an initial state for the DINING-PHILOSOPHERS system, given by a constant initial-state, with the strategy rewInv with the previous invariant as follows.

```
red rewInv(DINING-PHILOSOPHERS,
           DINING-PHILOSOPHERS-PL-PREDS,
           initial-state,
           deadlock-free) .
```

With this command, we execute the system by allowing the nondeterministic application of the rules in the module, but with the guarantee that the invariant is satisfied by all the states in the trace.

6 Concluding Remarks

We have proposed generic invariant-driven strategies, which control the execution of systems by guaranteeing that the given invariants are satisfied. Our strategies are generic in the sense that they are parameterized by the system whose execution they control, by the logic in which the invariants are expressed, and by the invariants themselves. This parameterization, together with the level of modularization of the approach, allows improving quality factors such as extensibility, understandability, usability, or maintainability. We have illustrated its use with invariants expressed in propositional calculus. However, the good properties of Maude as a logical and semantic framework [8], in which many different logics and formalisms can be expressed and executed, allow us to use other logics as parameters of our strategies.

The strategy rewInv given in Section 4, although valid for logics like propositional logic, has to be slightly modified in the case of logics like temporal logics. We have already experimented with future time linear temporal logic (LTL for short). In this case, the satisfaction of LTL formulae cannot be decided considering particular states, but we need to look at complete traces. For example, consider the invariant restriction []P (P always holds). This invariant requires any future state to maintain the property P, and obviously this cannot be guaranteed just considering the actual state. Our approach to deal with temporal logic is based on the one proposed by Havelund and Roşu in [6] for monitoring Java programs, based on the progressive transformation of the invariant restrictions when the system state evolves, possibly obtaining a new invariant when the system state changes.

References

[1] P. Borovanský, H. Cirstea, H. Dubois, C. Kirchner, H. Kirchner, P.-E. Moreau, C. Ringeissen, and M. Vittek. ELAN v 3.3 user manual, Third edition. Technical report, INRIA Lorraine & LORIA, Nancy, France, Dec. 1998.

[2] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, (285):187–243, 2002.

[3] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. Maude 2.0 manual. Available in http://maude.cs.uiuc.edu., June 2003.

[4] M. Clavel, F. Durán, S. Eker, J. Meseguer, and M.-O. Stehr. Maude as a formal meta-tool. In J. Wing, J. Woodcock, and J. Davies, editors, *FM'99 - Formal Methods (Vol. II)*, volume 1709 of *Lecture Notes in Computer Science*, pages 1684–1704. Springer, 1999.

[5] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In J. Goguen and G. Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Action*. Kluwer, 2000.

[6] K. Havelund and G. Roşu. Rewriting-based techniques for runtime verification. To appear in Journal of Automated Software Engineering.

[7] J. Hsiang. Refutational theorem proving using term rewriting systems. *Artificial Intelligence*, (25):255–300, 1985.

[8] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. volume 9, pages 1–87. Kluwer Academic Publishers, second edition, 2002.

[9] T. McComb and G. Smith. Animation of Object-Z specifications using a Z animator. In *First International Conference on Software Engineering and Formal Methods (SEFM'03)*, 2003.

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

[11] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1998.

[12] M. Utting. The Jaza animator. The system and its documentation is available at http://www.cs.waikato.ac.nz/~marku/jaza/.

[13] J. Warmer and A. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley, 2003.