Specification and prototyping of network protocols in rewriting logic*

Peter Csaba Ölveczky
Computer Science Laboratory, SRI International, Menlo Park, U.S.A.
and
Department of Informatics, University of Bergen, Norway

Sigurd Meldal
Computer Science Department, Cal Poly, San Luis Obispo, U.S.A.

1 Introduction

The size and complexity of modern software systems make it almost impossible to avoid errors in their requirements and design specifications. A software engineer needs all the tools and help he can get, and still the engineering process is error-prone and difficult. Formal methods have been suggested as an efficient and effective approach to error reduction during the software engineering process. However: "Software errors are ultimately caused by human errors: typing errors, errors of understanding, errors of thought. It is just as likely for the engineer writing the formal specification to make a mistake as it is for a programmer to do so." [12]. In fact, it seems that errors in *specifications* occur at least as frequently as errors in program code, when measured in terms of defects per line of code/specification. For example, of the 197 critical defects identified during integration and system testing of the Voyager and Galileo spacecraft, only 3 were due to coding errors [35, 21].

A majority of defects can usually be traced back to errors in requirements, interfaces, and intrinsic difficulties of the problem domain [35]. For the spacecraft data above, approximately 50% of the defects were traced to requirements (mainly omissions), and 25% to each of interfaces and design.

Not only are defects most likely to be introduced in the early stage of software development, defects introduced at that stage are also the most expensive to correct. Boehm [2] estimates that specification defects are about 100 times more expensive to correct than are implementation errors. Consequently, tools that can improve the early requirements, specification and design phases are likely to reward the practitioner with a handsome pay-off.

*Supported by DARPA through Rome Laboratories Contract F30602-97-C-0312, by Office of Naval Research Contracts N00014-95-C-0225 and N00014-96-C-0114, by National Science Foundation Grant CCR-9633363, and by NFR under contract 100426/410.

The successful application of tools to the early stages of software development requires an increase in specification rigor. The better the semantics of the specification formalism is defined, the more likely it is that tools can be created that can provide insights into the consequences of particular specification and design choices.

The use of formal specification methods (i. e., methods derived from mathematical logic for the specification of computational systems) give unambiguous systems specifications that may be subjected to mathematical verification of their correctness with respect to precisely stated requirements. Formal specifications are also likely to give specifications at a reasonable level of abstraction.

Though formal methods have not quite lived up to the full potential envisioned by early proponents, their usefulness, in particular for developing safety-critical systems applications, is gradually being realized in industry (see e.g . [13]). For instance, SRI's PVS system could have detected the "Pentium bug" during routine formal verification [33]. Moreover, it is unlikely that standard testing methods would have identified the defect [30]. Other industrial formal verification projects using PVS discovered subtle defects in Rockwell's AAAMP5 microprocessor [38] and the possibility of firing a failed jet in a space shuttle [34]. Some U.S. government agencies now *require* the use of formal methods for validation and specification of safety-critical system components [13].

The focus for formal methods has traditionally been the use of *proofs* to verify the correctness of specifications and conformance of purported implementations. Complementing this approach, a recent research trend has been to investigate other uses of formalized specifications: automatic conformance testing (dating back to the work on Anna [20], recently developing architecture conformance testing tools [22]), semantic analysis (such as advanced type checking), and (semi-)automatic synthesis of executable prototypes. The focus of this article is the latter.

1.1 Executable specifications

Algebraic specification techniques, λ-calculi, and logic programming languages have been used to define executable formal specifications of the functional aspects of software systems. These formalisms are *function-oriented* (or relational); they specify some function (relation) which produces some output value given some input. This function-oriented setting, relating input to consequent output upon termination of the system, is not particularly well-suited to modeling nondeterministic, concurrent, reactive, and/or distributed systems. In fact, a reactive system, such as an operating system, is often not intended to terminate at all, and modeling such a system by a function-oriented input-output relation is somewhat counter-intuitive. Instead, it is better to take into account the fact that systems interact with other systems and hence may exhibit patterns of stimulus/response relationships that vary over time [8].

There are a number of operational models of concurrency, such as various extensions of labeled transition systems, where transitions are labeled with actions representing interaction with the environment. The actions are *atomic*, allowing us to focus on intrinsic properties of concurrency by abstracting away from the structures of the actions themselves. Concurrency and communication are introduced into these

models by adding constructs that model particular kinds of communication as language primitives. Labeled transition systems are often studied in the context of *process algebras* such as CCS [29], ACP [1], and CSP [14]. Petri nets [31] extend this state/transition setting by allowing states to be distributed across different locations. These formalisms must be extended with facilities for specifying data types in specification languages such as e. g., LOTOS [16, 3], OCCAM [15], and colored and algebraic Petri nets [17, 32] to allow more complex data structures in the systems.

Executable specifications of reactive systems may be interpreted (and thus executed) in various ways, e.g., following *all* possible execution traces, simulating *one* run, etc. The operational choice of execution model is usually either embedded in the language semantics, or defined by an extra-logical set of strategy constructors.

Rewriting logic [24, 25, 26, 28] extends algebraic specification techniques to concurrent and reactive systems. Among its possible advantages over other executable specification formalisms are its being based upon a fairly natural and well-known formalism, its integration of static and dynamic system aspects, the abstract modeling of communication, and its allowing user-defined strategies specified *in the logic itself*. Rewriting logic seems particularly suitable for specifying security and communication protocols (e.g., see [10, 18]). Such protocols are complex enough to warrant prototyping and their operational nature fits very well with rewriting logic.

In this article we explore the utility of rewriting logic through the specification of a new protocol for reliable broadcasting in dynamic networks. The specification of the protocol originated as an informal specification suggested by José Garcia-Luna [11]. The goal of the protocol is to improve on existing protocols for efficient and reliable broadcasting of messages in dynamic networks. The desire is to arrive at a readable and understandable formal specification, to validate the protocol by executing the specification, and to prove that it satisfies certain properties. Other members of the group developing and specifying the protocol are Grit Denker, José Meseguer, Brad Smith, and Carolyn Talcott.

The paper is organized as follows: Rewriting logic and its features supporting object-oriented specifications are briefly introduced in Section 2. Section 3 outlines the specification of a version of the communication protocol for static networks and discusses the role of rewriting logic in the specification process. Section 4 touches upon verification issues, and Section 5 sums up our experience using rewriting logic for this specification effort. There are two appendices that are made available at <http://www.csc.calpoly.edu/~smeldal/NIK98/NIK98App.pdf>.

2 Rewriting logic

Rewriting logic [24, 25, 26] extends algebraic specification techniques to handle nondeterministic, reactive, and concurrent systems. It is based on the following two observations:

1. Term rewriting is intrinsically concurrent: if $f(x, y) \longrightarrow g(x, y)$, $a \longrightarrow a'$, and $b \longrightarrow b'$ in one step, then the term $f(a, b)$ may rewrite to $g(a', b')$ and $f(a', b')$ in one step as well.

2. Rewriting is not necessarily just the operational counterpart of equational deduction, but can be used to specify *change* in a system, and therefore, also non-confluent[1] and non-terminating rewrite systems are meaningful.

Rewriting logic is *a logic of change* where the dynamic parts of the system are specified by labeled conditional rewrite rules and the static and structural parts are specified by equations. A rewrite theory $\mathcal{R}$ is a tuple (Σ, E, L, R) where (Σ, E) is an *equational specification* (be it unsorted, many-sorted, order-sorted, membership, ...), L a set of *labels*, and R a set of *rewrite rules*. Sentences are of the form

$$[t]_E \longrightarrow [t']_E$$

the intended meaning being that a state $[t]_E$ of a system *could* change to a state $[t']_E$ by repeated application of the following deduction rules[2] [24]:

Reflexivity. For each $[t] \in \mathcal{T}_{\Sigma,E}(X)$,

$$\frac{}{[t] \longrightarrow [t]}$$

Congruence. For each $f \in \Sigma_n, n \in \mathcal{N}$,

$$\frac{[t_1] \longrightarrow [t'_1] \quad \ldots \quad [t_n] \longrightarrow [t'_n]}{[f(t_1, \ldots, t_n)] \longrightarrow [f(t'_1, \ldots, t'_n)]}$$

Unconditional replacement. For each $l : [t(x_1, \ldots, x_n)] \longrightarrow [t'(x_1, \ldots, x_n)]$ in R,

$$\frac{[w_1] \longrightarrow [w'_1] \quad \ldots \quad [w_n] \longrightarrow [w'_n]}{[t(w_1/x_1, \ldots, w_n/x_n)] \longrightarrow [t'(w'_1/x_1, \ldots, w'_n/x_n)]}$$

Transitivity.

$$\frac{[t_1] \longrightarrow [t_2] \quad [t_2] \longrightarrow [t_3]}{[t_1] \longrightarrow [t_3]}$$

where $t(u_1/x_1, \ldots, u_n/x_n)$ denotes the simultaneous substitution of u_i for x_i in t for every $i \in \{1, .., n\}$. Intuitively, the entailment relation $\longrightarrow$ is the reflexive and transitive closure of the "one-step rewrite modulo the set E of equations" relation. A specification is operational if E is a set $E' \uplus Ax$ where E' is a set of (oriented) equations that is Church-Rosser and terminating modulo the set Ax of structural axioms, and where the specification is coherent. In order to simplify the presentation, we shall omit the equivalence class notation for terms and simply write t for $[t]_E$.

The models of rewriting logic are categories, and the initial model is the category where objects are E-equivalence classes of ground terms. There is an arrow from $[t]_E$ to $[t']_E$ iff there is a rewrite from $[t]_E$ to $[t']_E$. It has been demonstrated that models of reactive and concurrent systems such as Petri nets, CCS, Actors, and

[1] A system is *confluent* if, for any starting state, there is only one possible termination state

[2] We give the rules for the unsorted unconditional case; for *conditional* rewrite rules, see [24].

UNITY, can be expressed in a natural way in rewriting logic [24, 26]. As a matter of fact, rewriting logic is employed to state the semantics for Actors [39].

The language Maude [5, 25] is based on rewriting logic and membership equational logic (an extension of order-sorted algebra) [27]. The current Maude implementation at SRI International can reach up to 590.000 rewrites per second on applications running on a 200MHz Pentium Pro.

Although Maude is distributed with a default interpreter, the user may take control of the way in which the rules are applied; indeed, if a specification is non-confluent or non-terminating the user not only *wants* to control the application of the rules, but he *needs* to be able to control it. Instead of providing the user with extra-logical control commands to define the control strategy, Maude uses the fact that rewriting logic is *reflective* to allow the user to define his own strategies *within rewriting logic itself*[3] [4, 6, 7]. Standard strategies (such as exploring *all* rewrite paths, show all irreducible forms, etc.) can be retained in libraries and thus be reused or extended with particular strategies for given applications.

2.1 Object-oriented specifications in Maude

Rewriting logic has proven particularly useful when specifying concurrent systems in an object-oriented style – a style which seems very appropriate for the specification of communication protocols.

In Maude, a class declaration

```
class C | att1: s1, ... , attn: sn .
```

defines a class `C` with attributes `att1` to `attn` of sorts `s1` to `sn`. The declaration is equivalent[4] to an operator declaration[5]

```
op < _:C | att1: _, ... , attn:_> : Oid s1 ... sn -> Object .
```

where `Oid` is the sort of object names. A term `<O:C | att1: val1, ... , attn: valn>` denotes an object `O` of class `C` with attribute values `val1` to `valn`.

A *message* is a term of the sort `Message`, and the global state, or *configuration*, is a multiset of objects and messages, with multiset union defined by juxtaposition:

[3] Rewriting logic is reflective in the precise sense that there is a finitely presented "universal" theory U such that for any finitely presented rewrite theory T we have the following equivalence

$$T \vdash [t]_{E_T} \longrightarrow [t']_{E_T} \iff U \vdash [\langle \overline{T}, \overline{t} \rangle]_{E_U} \longrightarrow [\langle \overline{T}, \overline{t'} \rangle]_{E_U}$$

where $\overline{T}$ and $\overline{t}$ are terms representing T and t as data elements of U. Key features of the universal theory U are provided in the built-in Maude module `META-LEVEL` which can be extended in a completely user-definable way to specify strategies controlling the rewriting in a way *internal* to rewriting logic itself.

[4] We consider the setting without inheritance. Inheritance is treated in [25].

[5] The symbol _ marks the place for the arguments of a "mix-fix" operator symbol.

```
subsort Object Message < Configuration .
op empty : -> Configuration .
op _ _ : Configuration Configuration -> Configuration
         [assoc comm id: empty] .
```

Rewriting in configurations is rewriting modulo associativity, commutativity, and identity, i. e., multiset rewriting.

The dynamic behavior of concurrent object systems is axiomatized by specifying each of its concurrent transaction patterns by a labeled rewrite rule. For example, the rule

```
rl [rule1] :
   (m1)<O:C|a1:x, a2:y, a3:z> => <O:C|a1:x+y, a2:y, a3:z>(m2) .
```

defines a (family of) transition(s) in which a message `m1` is consumed by an object `O` of class `C`, with the effect of altering the attribute `a1` of the object and of generating a new message `m2`. By convention, attributes whose values do not change and do not affect the next state of other attributes need not be mentioned in a rule. Thus the above rule could also be written

```
rl [rule1] : (m1)<O:C|a1:x, a2:y> => <O:C|a1:x+y>(m2) .
```

In this paper, we given an abbreviated example of how one may use Maude's object-oriented features, its underlying formal semantics, and its rewrite engine to specify, validate, and analyze a communication protocol.

3 Specifying a communication protocol in Maude

3.1 Broadcasting in networks

Not much work exists on reliable broadcast protocols in dynamic networks. Most broadcast protocols are based on the PI and PIF protocols [37, 36], and all the broadcasting protocols for dynamic topologies extending these are based on the routing protocol by Merlin and Segall [23] which incurs too much communication to be attractive for, say, a wireless network [11]. It is the long term goal of our work to define a more general approach (in terms of mobility of nodes) and more efficient protocols for reliable broadcasting in dynamic networks. The starting point of our work was an informal draft specification of the protocol given by José Garcia-Luna [11], and a group consisting of Garcia-Luna, Grit Denker, José Meseguer, Brad Smith, Carolyn Talcott, and the first author are currently working on specifying the protocol.

An important subcase of the general, dynamic network topology is the static setting. This paper reports on the specification of the protocol, the *reliable broadcast protocol (RBP)*, for the static topology. A static network is modeled as an undirected connected graph where each node knows its network but does not know anything more about the topology. The objective of the protocol is to broadcast the message

from a source node to all the nodes in the network and to give feedback to the source.

We are currently working on defining the protocol for dynamic topologies, and the first draft proposal is directly based on the specification presented below.

3.2 The specification process

There are many fuzzy requirements and much implicit knowledge in most informal specifications, such as in the starting point of our specification effort, a specification written in natural language and pseudo-code. The *formalization* part of the specification process aims at clarifying fuzzy requirements and making essential, implicit knowledge explicit. In our case, the fact that each channel between two neighbors preserves the message order was an example of essential, but unstated, information.

Besides the proof oriented methods of classical formal methods, specifications written in rewriting logic may also be validated immediately through their execution, bringing to bear the methodologies of standard testing procedures. This prototyping possibility comes for free, and we used this feature every time a specification was modified, and often encountered counterexamples immediately on even quite simple test examples (such as a network of three nodes). In the work reported here we used Maude's default interpreter (which simulated an arbitrary run of the protocol), but as the experiments become more sophisticated we expect to have to modify this default behavior.

The explorations through execution helped eliminate errors of syntax and of thought; furthermore, the built-in Maude facilities for tracing an execution were useful for discovering where and why errors occurred.

After test executions the protocol was subjected to close *analysis* by using Maude's meta-programming capabilities to define a strategy which gave the final state of every possible execution scenario. For non-terminating systems, this setting could be modified to give e. g., every state which is reachable in less than 50 one-step rewrites from some initial state. Executing the system with this exploratory strategy, we discovered a deadlock scenario which we hadn't considered and which invalidated the ideas underlying the protocol.

With Maude's meta-programming features, the user may define the rewrite strategies himself, enabling the analysis of a specification under various scenarios.

Based on the information extracted from the exploratory analysis, our group identified and improved solution for the static network setting, which is presented next.

3.3 The protocol

In this section we outline a Maude specification of the modified protocol for the static topology, one of the preliminary results of our specification work.

For each run of the protocol there is a single source node, and there is one message that needs to be sent to all the other nodes in the network. (Handling multiple

messages from each source can be accommodated easily by means of bit vectors or counters.) The protocol proceeds roughly as follows:

1. The source node sends the message to all its neighbors.
2. When a node n receives the message for the first time, it remembers the sender (the *parent* of n), and sends the message to all its neighbors except its parent.
3. When a node n receives a message it has already seen, the node sends an acknowledgment to the sender of this latest message (its *sibling*).
4. When a node has received an acknowledgment from all its neighbors except its parent, it sends an acknowledgment to the parent and its cycle is finished.
5. The protocol terminates when the source has received acknowledgments from all its neighbors.

The protocol is similar to the well-known PIF protocol for static networks, but incurs a few more messages. In comparison with PIF-based protocols, the innovation of this protocol is the extra set of acknowledgment messages between "sibling" nodes. This adds to the complexity of the protocol, but our expectation is that this redundancy will result in an improved protocol for dynamic networks, better than the existing extensions of the PIF protocol.

We need to provide some application-specific data types such as sets of object identifiers and multiple copies of a certain message. First, we define object identifiers with two special object values `none` and `source`:

```
sorts  Oid DefOid SrcOid DefSrcOid .
subsort Oid < DefOid .              *** Oid with default value
subsort Oid < SrcOid .              *** Oid with value source
subsort SrcOid < DefSrcOid .
subsort DefOid < DefSrcOid .
ops a b c d e f : -> Oid .          *** Six object names used.
op none : -> DefOid .
op source : -> SrcOid .
```

Sets of `Oid`s are multisets with an additional idempotency axiom:

```
sort OidSet .
subsort Oid < OidSet .
op nil : -> OidSet .
op _ _ : OidSet OidSet -> OidSet [assoc comm id: nil] .
var A : Oid .
eq A A = A .
```

together with the usual functions on sets. The class `Node` is declared

```
class Node | nbs: OidSet, parent: DefSrcOid,
             recdMsg: OidSet, recdAck: OidSet .
```

where `nbs` denotes the set of neighbor objects, `parent` the node's parent (or the value `none` if it has no parent and `source` if the node is the source node for the cycle), and `recdMsg` and `recdAck` the sets of nodes from which the objects has received messages and acknowledgments respectively.

The set of messages is

```
op (to _ Broadcast _) : Oid MessageContent -> Message .
op (msg _ From _ To _) : MessageContent Oid Oid -> Message .
op (ackParent _ From _ To _) : MessageContent Oid Oid -> Message .
op (ack _ From _ To _) : MessageContent Oid Oid -> Message .
```

where `(to A Broadcast m)` indicates that node `A` is the source of the broadcasting cycle of the message with content `m`. The message `(msg m From A to B)` represents a message with content `m` sent from `A` to `B`, a message `(ackParent m From A to B)` represents an acknowledgement from `A` to its parent `B`, and `ack m From A to B` represents an acknowledgment from `A` to its sibling `B`[6].

A node needs to send a message to all its neighbors except the parent at certain stages of the protocol. At first one might not recognize such a simple fact, and model the broadcast to the neighbors as a separate message sent to each. Upon closer inspection, it is natural to consider this a single event. Transforming a set of events into a single, more abstract one, can be quite tedious in process algebras. In rewriting logic it turns out to be quite straight-forward to add a new feature of "multimessages" to the specification. A message

```
(multimsg m From A To N)
```

for `N` a set `B1, B2, ..., Bn` of objects can simply be defined as denoting the construct

```
(msg m From A to B1) (msg m From A to B2) ... (msg m From A to Bn)
```

Formally:

```
vars A B : Oid .   var N : OidSet .   var M : MessageContent .
eq (multimsg M From A To B N) = (msg M From A to B)
                                (multimsg M From A to N-B) .
eq (multimsg M From A To nil) = empty .
```

Given the variable declarations

```
vars A B C : Oid .  vars N N' N'' : OidSet .
var M : Message .   var D : DefOid .
var S : SrcOid .    var O : DefSrcOid .
```

the transitions in the system can be defined as follows:

[6]The division into `ack` and `ackParent` messages can be seen as an abstraction from sequence numbers in a channel or other means of enforcing the necessary message-order.

```
rl [Start] : (to A Broadcast M) <A:Node|nbs: N, parent: none> =>
             <A:Node|parent: source> (multimsg M From A To N) .
```

This rule starts the cycle by letting the source node send a message to all its neighbors.

```
rl [RecMsg1] : (msg M From A To B) <B:Node|nbs: N, parent: none> =>
               <B:Node|parent: A> (multimsg M From B To N-A) .
```

This rule only applies to objects whose parent attribute has the value `none`, i.e., this would be the first time object `B` received the message, which it then propagates to all its neighbors except the parent.

```
rl [RecMsg2] :
   (msg M From A To B) <B:Node|nbs: N, parent: S, recdMsg: N'> =>
   <B:Node|recdMsg: N' A> (ack M From B To A) .
```

Due to the way the sorts are defined, `S` cannot represent the value `none`. Consequently, `B` has a parent and therefore has already seen the message – `B` should not propagate the message further, but should instead send an acknowledgement to the sender.

```
rl [RecAckSibling] :
   (ack M From A To B) <B:Node| recdMsg: A N, recdAck: N'> =>
   <B:Node|recdAck: N' A> .
```

An acknowledgment from a sibling is not consumed unless the node `B` already has received a message from the sibling. This restriction on the consumption of `ack`-messages is justified by the assumption that channels preserve the order of messages.

```
rl [ackParent] :
   <A:Node|nbs: B N, parent: B, recdMsg: N', recdAck: N> =>
   <A:Node|parent: none, recdMsg: nil, recdAck: nil>
      (ackParent M From A To B) .
```

Node `A` has received an acknowledgment from all neighbors except the parent and can acknowledge the parent and "reset" its attributes for a next cycle.

```
rl [RecAckParent] :
   (ackParent M From A To B) <B:Node|recdAck: N> =>
   <B:Node|recdAck: N A> .
```

The node just records the reception of an acknowledgment message.

```
rl [ResetSource] :
   <A:Node|nbs: N, parent: source, recdMsg: N', recdAck: N> =>
   <A:Node|parent: none, recdMsg: nil, recdAck: nil> .
```

The source has received an acknowledgment from all neighbors, and a cycle of the protocol terminates, while the source resets its attributes.

The initial state is defined as a graph, i.e., a set of objects, each identifying its neighbors. An initial configuration with four nodes `a`, `b`, `c`, and `d`, channels `(a,b)`, `(a,c)`, `(b,c)`, and `(c,d)`, and source `a`:

```
eq test4 =
     (to a Broadcast m)
     <a:Node|nbs: b c, parent: none, recdMsg: nil, recdAck: nil>
     <b:Node|nbs: a c, parent: none, recdMsg: nil, recdAck: nil>
     <c:Node|nbs: a b d, parent: none, recdMsg: nil, recdAck: nil>
     <d:Node|nbs: d, parent: none, recdMsg: nil, recdAck: nil> .
```

Executing the specification with `test4` as an initial term should end with an irreducible state which equals the initial state minus the broadcast message which triggers the cycle.

Exploratory analysis of all execution scenarios in the system has given the hoped-for results for the networks with a few nodes. Given the exponential explosion of possible runs with increases in network sizes, the system gets mired even for fairly small networks when called upon to perform an exhaustive exploration of possible runs. On our platform it failed to terminate within a reasonable time for a network of 6 nodes. Apart from the exponential number of runs, this fact could also be attributed to non-optimal coding of the strategy expression for finding all rewrites, which emphasized readability over efficiency.

4 Verification

Rewriting logic is a logic that axiomatizes the computational aspects of a concurrent system. In fact, deduction in rewriting logic corresponds exactly to concurrent computation in the system. A specification in rewriting logic may therefore be considered a *prescriptive* or *operational* description of a system.

In contrast, temporal logic takes a *descriptive* view of the system, making assertions about its global safety and liveness properties such as "the message content will not be corrupted" and "every node will receive the message within finite time." Both kinds of specification are useful for the specification of the protocol. Clearly, saying that every node will see the message does not specify the protocol at a level useful for guiding later realization, but states global requirements the system must satisfy. In particular, for dynamic networks, a temporal formula could express *which* nodes in the network are required to see the message.

The *verification* part of the specification process consists of proving that the initial model $\mathcal{T}_{\mathcal{R}}$ (viewed as a Kripke structure) satisfies the temporal requirements. Finding the most suitable temporal logics for expressing global properties about (object-oriented) rewrite specifications and giving proof techniques for proving that a rewrite theory satisfies the global requirements are interesting and open research topics. Work exploring these topics includes Lechner's adaption of μ-calculus [19]

and Denker's version of the object-oriented Distributed Temporal Logic (DTL+) [9].

5 Summary

We have illustrated how one may use rewriting logic and Maude to formally specify communication protocols. The specification was tested by executing it on chosen test states using Maude's default interpreter, and was analyzed by finding all non-rewritable reachable states from these initial test configurations. The specification has not been verified wrt. global requirements.

Rewriting logic specifies the protocol on an intermediate level, bridging the specification gap between a descriptive requirements specification and an actual implementation of the protocol on a distributed network. The rapid prototyping of a specification by means of its interpretation by Maude adds a significant tool to the toolkit available to a software engineer using formal specification methods. From a methodological point of view, this option of immediate execution to perform a "model checking"-style analysis on the specification seems quite advantageous when checking for defects early in the specification process, before proofs of properties are attempted.

We believe that a rewriting logic specification allows the capture of the essential operational aspects of the protocol at a very natural level, while it abstracts away from particular communication models. The advantage of the more abstract communication model was indicated by the smooth integration of the new "multimessage" construct in the specification.

Furthermore, while formalisms based on process algebra often have a fixed set of strategies external to the logic, strategies are completely user-definable *within* the logic using Maude's meta-programming capabilities. Although we have not made extensive use of this flexibility so far, more complex strategies will be needed for dynamic networks. Strategies stating e. g. that every 20th time a link *could* fail, it will fail, seem difficult to express in other languages without interfering with the systems specification itself, while such a property may be stated at the rewriting logic meta-level without altering the (object-level) specification itself.

Future work includes specifying the protocol for dynamic networks and integrating real-time aspects such as message time-outs, as well as finding the appropriate temporal logics and proof techniques for verifying object-oriented rewrite specifications.

Acknowledgments

The specification effort upon which this paper is based is joint work with Grit Denker, José Garcia-Luna, José Meseguer, Brad Smith, and Carolyn Talcott. We thank all of them, and in particular José Meseguer for very helpful discussions on specification and rewriting logic. We also thank Narciso Martí-Oliet for many helpful comments on earlier versions of this paper.

References

[1] J. C. M. Baeten and W. P. Weijland. *Process Algebra.* Cambridge University Press, 1990.

[2] B. W. Boehm. *Software Engineering Economics.* Prentice-Hall, 1981.

[3] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14, 1987.

[4] M. G. Clavel. *Reflection in General Logics, Rewriting Logic, and Maude.* PhD thesis, University of Navarre, 1998.

[5] M. G. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Proc. 1st Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science.* Elsevier, 1996.

[6] M. G. Clavel and J. Meseguer. Axiomatizing reflective logics and languages. In G. Kiczales, editor, *Proc. Reflection'96.* Xerox Parc, 1996.

[7] M. G. Clavel and J. Meseguer. Reflection and strategies in rewriting logic. In J. Meseguer, editor, *Proc. 1st Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science.* Elsevier, 1996.

[8] R. Cleaveland and S. A. Smolka et al. Strategic directions in concurrency research. *ACM Computing Surveys*, 1996.

[9] G. Denker. From rewrite theories to temporal logic theories: A distributed temporal logic extension of rewriting logic. Manuscript, SRI International, 1998.

[10] G. Denker, J. Meseguer, and C. Talcott. Protocol Specification and Analysis in Maude. In N. Heintze and J. Wing, editors, *Proc. Workshop on Formal Methods and Security Protocols, 25 June 1998, Indianapolis, Indiana*, 1998.

[11] J. J. Garcia-Luna. Reliable broadcasting in computer networks, 1998. Manuscript, UC Santa Cruz.

[12] V. Hamilton. The use of Z within a safety-critical software system, 1995. In [13].

[13] M. G. Hinchey and J. P. Bowen (eds). *Applications of Formal Methods.* Prentice-Hall, 1995.

[14] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[15] Inmos International. *OCCAM-2 Reference Manual.* Prentice-Hall, 1988.

[16] Information systems processing – open systems interconnection – LOTOS. Tech. rep. International Standards Organization DIS 8807, 1987.

[17] K. Jensen. *Coloured Petri Nets, Basic Concepts, Analysis Methods and Practical Use*, volume 1 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1992.

[18] S. Krogdahl and O. Lysne. Verifying a distributed list system: A case history. *Formal Aspects of Computing*, 3, 1997.

[19] U. Lechner. *Object-Oriented Specification of Distributed Systems*. PhD thesis, University of Passau, 1997. Available at: `www.mcm.unisg.ch/~ulechner`.

[20] D. C. Luckham and F. W. von Henke. An overview of Anna, a specification language for Ada. *IEEE Software*, 2(2):9–23, March 1985.

[21] R. R. Lutz. Analyzing software requirements errors in safety-critical embedded systems. In *IEEE International Symposium on Requirements Engineering*, 1993.

[22] S. Meldal and D. C. Luckham. NSA's MISSI reference architecture – moving from prose to precise specification. In M. Broy and B. Rumpe, editors, *Requirements Targeting Software and Systems Engineering*, volume (TBD) of *Lecture Notes in Computer Science*, pages 293–329, 1998.

[23] P. M. Merlin and A. Segall. A failsafe distributed routing protocol. *IEEE Trans. Commun.*, 27(9):1280–1288, 1979.

[24] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96, 1992.

[25] J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*. MIT Press, 1993.

[26] J. Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In *Concur'96*, volume 1119 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[27] J. Meseguer. Membership algebra as a semantic framework for equational specification. In F. Parisi-Presicce, editor, *Proc. WADT'97*, volume 1376 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[28] J. Meseguer. Research directions in rewriting logic. In U. Berger and H. Schwichtenberg, editors, *Computational Logic, NATO Advanced Study Institute, Marktoberdorf, Germany, July 29 – August 6, 1997*. Springer-Verlag, 1998. To appear.

[29] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[30] V. Pratt. Anatomy of the Pentium Bug. In P. D. Mosses, M. Nielsen, and H. I. Schwartzbach, editors, *TAPSOFT'95: Theory and Practice of Software Development*, volume 915 of *Lecture Notes in Computer Science*, pages 97–107. Springer-Verlag, 1995.

[31] W. Reisig. *Petri Nets*, volume 4 of *EATCS monographs on Theoretical Computer Science*. Springer-Verlag, 1985.

[32] W. Reisig. Petri nets and algebraic specifications. *Theoretical Computer Science*, 41, 1991.

[33] H. Rueß, N. Shankar, and M. K. Srivas. Modular verification of SRT division. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, number 1102 in Lecture Notes in Computer Science, pages 123–134. Springer-Verlag, 1996.

[34] John Rushby. Automated deduction and formal methods. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, number 1102 in Lecture Notes in Computer Science, pages 169–183. Springer-Verlag, 1996.

[35] John Rushby. Mechanized formal methods: Progress and prospects. In *16th Conference on the Foundations of Software Technology and Theoretical Computer Science*, volume 1180 of *Lecture Notes in Computer Science*, pages 43–51. Springer-Verlag, December 1996.

[36] A. Segall. Distributed network protocols. *IEEE Trans. Info. Theory*, 29(1):25–35, 1983.

[37] A. Segall and B. Awerbuch. A reliable broadcast protocol. *IEEE Trans. Commun.*, 31(7):896–901, 1983.

[38] M. K. Srivas and S. P. Miller. Formal verification of the aamp5 microprocessor. In [13], 1995.

[39] C. L. Talcott. An actor rewriting theory. In J. Meseguer, editor, *Proc. 1st Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996.