Formal Models of Distributed Interoperation and Reflection

José Meseguer

Computer Science Department
University of Illinois at Urbana-Champaign

Motivation

A logical framework can be very useful as a way of achieving fundamental advances in system interoperability because:

- If the framework is truly flexible and general, it will provide a precise semantics for the new models of interaction that are needed in next-generation middleware and mobile systems.
- Semantic requirements for interoperation can be specified in such a framework, and can be made part of the documentation and code of components.
- Specifications can be made mathematically precise and unambiguous.

Motivation (II)

- Furthermore, different forms of formal analysis and verification become possible, crucial means to achieve high-assurance about the correctness, security, and reliability of complex systems.

- If the framework is reflective, systematic and semantically rigorous methods for system adaptation can be designed, prototyped, and formally analyzed within the framework.

- If the framework is wide-spectrum, both specifications and code can be rigorously developed and be reasoned about within the framework, and software tools ensuring correct transformation of specifications into code can be designed and implemented.

The Rewriting Logic Framework

Rewriting logic (RWL) is a logical framework that meets the above requirements and seems well suited as a semantic framework for adaptive system interoperation.

There is a substantial body of concrete evidence suggesting that this is the case. Among the over 300 papers on rewriting logic already published, work particularly relevant includes:

- work by Talcott on the rewriting semantics of actors and on distributed components
- work by Denker, Talcott, and Meseguer on the rewriting logic semantics of the “onion skin” model of distributed reflection, supporting composable communication services

The Rewriting Logic Framework (II)

- work by Talcott, Venkatasubramanian, and Meseguer giving a rewriting semantics for the two-level actor model (TLAM) of distributed reflection

- work by Najm and Stefani on a rewriting logic semantics for the Reference Model for Open Distributed Processing (RM-ODP)

- work by Viry, Nakajima, Stehr, Sen, Martí-Oliet, and Thati on the rewriting logic semantics of different mobile calculi such as Milner's π-calculus, and the calculus of mobile ambients of Cardelli and Gordon

The Rewriting Logic Framework (III)

- work by Stehr, Talcott and Meseguer (in cooperation with Carl Gunter) on rewriting logic formal semantics and simulator for the PLAN mobile language for active networks
- wok by Van Baalen et al. on a rewriting logic specification and analysis of key fault-tolerant protocols in the DaAgents mobile agent system
- work by Durán, Eker, Lincoln, Vedejo, and Meseguer on the design and implementation of Mobile Maude; and
- work by Denker, Talcott, and Meseguer on the rewriting logic specification and formal analysis of cryptographic protocols.

The Rewriting Logic Framework (IV)

A feature of primary importance is the great flexibility and generality of rewriting logic to express different models of concurrency and interaction, including models of interoperation for systems of distributed components.

Both the distributed states and the local concurrent interactions in such systems can be naturally specified by rewrite theories, in which the interactions are described by rewrite rules.

An important advantage of rewriting logic specifications is that they are executable. Therefore, they provide an executable formal model of the specified system, that can be simulated, and can be analyzed and reasoned about using a variety of model checking and proof techniques.

The Rewriting Logic Framework (V)

Maude is a high-performance rewriting logic language and system that supports executable specification and programming of distributed object-based systems.

Executability can be exploited not just for formal specification, symbolic simulation, and formal analysis purposes: with appropriate compilation techniques rewrite rules can in fact be the code of parallel, distributed, and mobile systems that are then much easier to develop, debug, and analyze than similar systems built with conventional code.

The Mobile Maude language is an example of this wide-spectrum capability connecting specifications and mobile code.

The Rewriting Logic Framework (VI)

Another key feature of the rewriting logic framework is reflection. Intuitively, a logic is reflective if it can represent its metalevel at the object level in a sound and coherent way. Rewriting logic logic is reflective in this sense. Specifically, there is a finitely presented rewrite theory $\mathcal{U}$ that is universal in the sense that for any finitely presented rewrite theory $\mathcal{R}$ (including $\mathcal{U}$ itself) we have the following equivalence

$$\mathcal{R} \vdash t \to t' \;\Leftrightarrow\; \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \to \langle \overline{\mathcal{R}}, \overline{t'} \rangle,$$

where $\overline{\mathcal{R}}$ and $\overline{t}$ are terms representing $\mathcal{R}$ and t as data elements of $\mathcal{U}$. Since $\mathcal{U}$ is representable in itself, we can achieve a "reflective tower" with an arbitrary number of levels of reflection.

The Rewriting Logic Framework (VI)

Among the many applications of reflection relevant for this research we can mention:

- module composition and adaptation: modules are data that can be composed, adapted, and transformed within the logic, even dynamically
- mobile reflection: by reflection, mobile agents can carry their own code, execute remotely, and can dynamically adapt their own code to respond to changes or threats in their environment; and
- meta-reasoning: more powerful reasoning capabilities become possible by reflection, and it is easy to develop formal reasoning tools as theories within the framework.

Formal Modeling, Analysis, and Verification

The fact that rewriting logic specifications are executable allows us to have a flexible range of increasingly stronger formal methods, to which a system specification can be subjected. Maude and its formal tools can be used to support the following methods:

- formal specification, which results in an executable model which can be symbolically simulated;
- model-checking analysis, which can uncover subtle errors by considering all behaviors of a system from an initial state; and
- formal proof, to gain the highest assurance about critical properties, which can be assisted by formal tools.

Formal Modeling, Analysis, and Verification (II)

The above methodology is supported by the Maude formal tools. First of all, Maude itself is a very versatile formal tool supporting methods 1–2 through its default interpreter, and through its model checker for linear temporal logic.

The Maude environment of formal tools also includes:

- an inductive theorem prover
- a Church-Rosser checker
- a termination checker and Knuth-Bedix completion tool
- a coherent checker; and
- the Real-Time Maude tool.

The Onion Skin Model

We illustrate the use of rewriting logic as a logical framework by giving a rewriting semantics for Agha et al.'s onion skin model, in which each actor has a metaactor that defines the semantics of its primitive actions.

For example, a message send by the actor becomes a request to its metaactor to transmit the message.

Dually, messages sent to the actor are first received by its metaactor, giving the metaactor the capability to control the receive semantics.

The Onion Skin Model

An actor composed with its metaactor appears from the outside like a normal actor, which can be controlled by a further metalevel actor.

These metalevel layers suggest the onion skin analogy.

Different distributed services, such as encryption, fault-tolerance, and different QoS services, can be supported by different metaactor layers.

Formalizing the Onion Skin Model

Onion skin layers are formalized using tower objects, that implement individual layers of a tower of objects.

Objects at the bottom of a tower represent application objects. Each metaobject has as a subobject the object tower immediately below, called its base. A top-level object implements the default metaactor and enables communication with the environment.

More precisely there are three object classes that structure towers: `Top` (for top-level objects), `MetaTower` (for intermediate layer metaobjects), and `Tower` (containing both metaobjects and application objects).

Formalizing the Onion Skin Model (II)

The class `Tower` has two attributes: `in` and `out`:

- `in` is a list of messages to be delivered to the object;
- `out` is a list of outgoing requests for message transmittal and object creation.

Application objects at the bottom of the tower are assumed to have the general form

```
< o : BTC |  in: msgs, out: reqs, atts >
```

where `BTC` is a subclass of `Tower`, and `atts` are additional internal state attributes.

Formalizing the Onion Skin Model (III)

Application objects then have their own rules to consume messages, change their state, and request creation of new objects and transmittal of messages.

Such rules are different for each application, but they are assumed to follow the general pattern,

```
< o : BTC |  in: msgs . msgs', out: reqs, atts >
  =>
< o : BTC |  in: msgs', out: reqs . reqs', atts' >
```

Formalizing the Onion Skin Model (IV)

Intermediate metaobjects belong to the `MetaTower` class, a subclass of the class `Tower` having an additional attribute, `base`, whose value is a tower object. The relation between a metaobject and its subobject is the same at each level of the tower—the base-meta relation. Hence `base` refers to the subobject at the level immediately below.

Metaobjects in `MetaTower` have the general form,

```
< o : MTC | in: msgs, out: reqs,  base: tobj, atts >
```

where `MTC` is a subclass of `MetaTower`, `tobj` is a tower object and `atts` are additional internal state attributes.

Formalizing the Onion Skin Model (V)

Metaobjects have boundary-crossing rules of the general form

```
 < o : MTC | in: msgs . msgs', out: reqs,
       base: < o : TC | in: dmsgs, out: ureqs . ureqs', tatts >,
       atts>
   =>
 < o : MTC | in: msgs', out: reqs . reqs',
       base: < o : TC | in: dmsgs . dmsgs', out: ureqs', tatts >,
       atts' >
```

That is, a metaobject can remove messages from its `in` queue and/or requests from the `out` queue of its `base` subobject; and it can put requests in its own `out` queue and/or messages in the `in` queue of its `base` subobject.

Formalizing the Onion Skin Model (VI)

Top-level objects belong to the `Top` class, with just one attribute, `base`, whose value is a tower object. Top objects only have rules for communication with the environment.

```
[in]  o <- msg
  < o : Top | base: < o : TC | in: msgs, out: reqs,  tatts > >
     =>
  < o : Top | base: < o : TC | in: msgs . msg, out: reqs, tatts > >

[out]
  < o : Top | base: < o : TC | in: msgs, out: req . reqs, tatts > >
    =>
  < o : Top | base: < o : TC | in: msgs, out: reqs, tatts > >
  createConf(req)
```

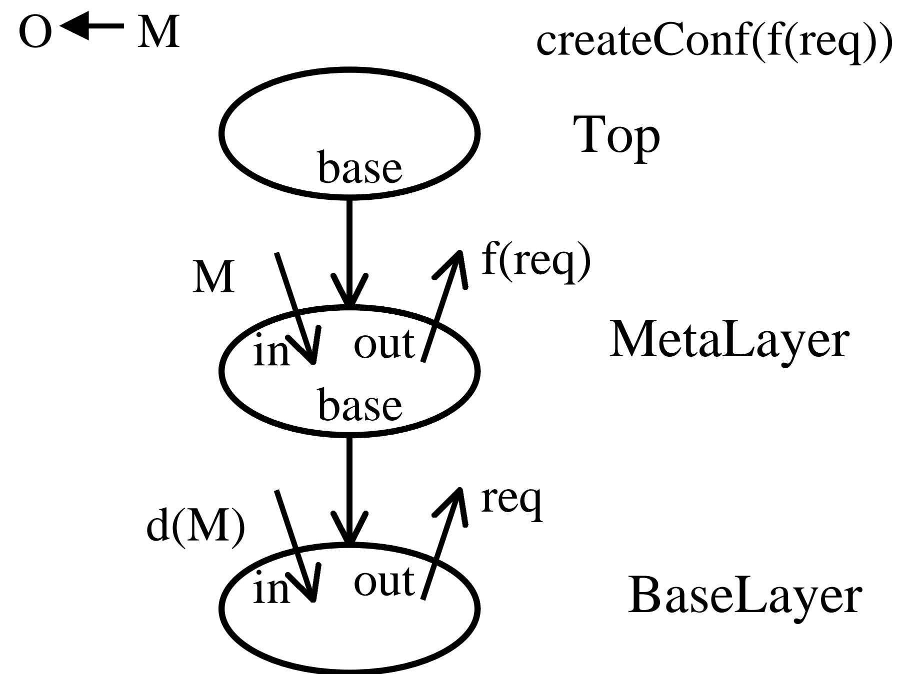

Figure 1: A metaobject tower