A Semantic Model for Safe Protocol Interaction

Sebastian Gutierrez-Nolasco,
Nalini Venkatasubramanian
School of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3430, USA
{seguti,nalini}@ics.uci.edu

Carolyn Talcott
Computer Science Laboratory
SRI International
Menlo-Park, CA 94025, USA
clt@cs.stanford.edu

ABSTRACT

Most communication subsystems support modular and reconfigurable communication protocols based on the Lego block model. In this model, complex protocols are built as collections of simpler protocols. However, protocol behavior is often expressed via informal descriptions and few work has been done to develop the underlying semantics that enables us to model and reason about protocol interactions. Without it, it is very difficult to identify critical properties that must be met for correct operation. In this paper we present a communication framework based on a semantic model of distributed object reflection and we illustrate how this model can be used to formalize and reason about interactions between communication protocols. We evaluate the overhead and feasibility of our approach by developing an abstract executable specification of the communication framework in Maude [3] and a Maude API to systematically translate Maude code into Java. This gives us two versions of our model to play with: an abstract one for analysis and a concrete one for real-world applications.

1. MODELING IN REWRITING LOGIC

In general, a rewrite theory is a triple $\mathcal{R} = (\Sigma, E, R)$ with (Σ, E) an equational specification with signature of operators Σ and a set of equational axioms E, and a collection of rewrite rules R. The equational specification describes the static structure of the configuration as an algebraic data type and is a purely functional part, while the dynamics is described by the rules in R representing local state transitions that can occur in the system axiomatized by $\mathcal{R}$ and that are applied modulo the equations E. In our model, a system is composed of two kinds of agents, base agents and meta agents. Base agents carry out application level computation, while meta agents are part of the run-time system, which controls the run-time behavior of the base level. Meta agents communicate with each other via message passing as do base agents, and they may also examine and modify the state of the base agents located on the same node. Base level agents and messages have associated run-time annotations that can be set and read by meta agents. The annotations are invisible to base level computation. Actions which result in a change of base level state, are called events and meta agents may react to them if they occur on their node. Thus, we can visualize the state of the system, often called a *configuration*, as a space in which objects and messages float and interact with each other following certain rules, which describe the behavior of individual objects. We model a configuration using the rewriting logic object model specified in [1].

SAC'06 April 23-27, 2006, Dijon, France

1.1 Components of the Model

Inspired by the original Russian Dolls model [2], objects are structured in nested configurations of meta objects that can control (base and meta) objects under them, called subobjects, with an arbitrary number of levels of nesting. This allows us to model processing nodes as meta objects that encapsulate agents (base and meta) and messages.

Base and meta agents: We model agents as objects with identity and state, configurations as soups of agent and messages, and agent behavior rules as rewrite rules. A base agent is represented as an object in some base agent class *BC* and each base agent has a meta agent, called messenger, which serves as the customized and transparent mail queue. Messages have the form *a* $<<$ *m*, where *a* is the agent identifier of the target agent and *m* is the message contents. Base agent rules are constrained to have the form

```
<a:BC|atts>, [a << m] => <a:BC'|atts'>, bconf if cond
```

where *BC* and *BC'* are base clases, *atts* and *atts'* are attribute value sets appropriate for the corresponding classes, *bconf* is a configuration of newly created base agents and messages, and [ ] indicates that the message part is optional. A meta agent is an object in some meta agent class *MC* and meta messages have the form *ma* $<<$ *mv*, where *ma* is the identifier of the meta agent.

Node: A node is an object of class *Node* and has an attribute *conf* whose value is a configuration containing meta agents and messages. Each node has three special meta agents: A base behavior task manager meta agent *btma* of class *BTMC* that represents and controls base execution behavior, and a communication meta agent *cma* of class *CMC* that represents and controls base communication semantics. These meta agents also implement the event handling semantics associated with base events. Finally, there is one communication manager in every node of the distributed system, which implements and controls protocols composition and interaction on that node. We expand the communication manager to encapsulate and control a set of available communication protocols *cp* and a pool of message coordinators agents *mc*, so it can coordinate message processing in the node.

Communication Manager: A meta object of class *CMgrC* has attributes *in, out, cpConf, conf*. The values of *in* and *out* are lists of incoming and outgoing messages of the form $(!msg^*)$, representing processed messages. The value of *cpConf* is the meta representation of the communication protocols currently loaded while the

value of *conf* is a configuration containing message coordinators and communication protocols.

```
<cmgr:CMgrC|in:imsg, out:omsg, cpConf:!cps,
      conf:<mc: MCC |...>, <cp: CPC |...>,mconf>
```

Message Coordinator: A meta object of class *MCC* has attributes *mailq, mpr, mrst, mip, ord, stage, spin*. The value of *mailq* is a list of incoming messages of the form *((!msg,!sndr,amap),!msl)* representing the request to process message *msg* sent by base agent *sndr* and message annotation map *amap* with the communication protocols specified in *!msl*. The values of *mpr*, *mrst* and *mip* are meta representations of the master prerequisites, restrictions and interaction parameters (respectively) of the communication protocols specified in *!msl*, while the value of *ordl* is the meta representation of the composition order that needs to be obeyed to ensure correct composition of communication protocols. Finally, the value of *stage* is the meta representation of the sequence of messages that a protocol has received in the current run.

```
<mc:MCC|mailq:msg, pr:prlst, rst:rstlst, ip:iplst,
    ord:ordl, stage:acstage>
```

Communication Protocol: A meta object of class *CPC* has attributes *sndq, rcvq, pr, rst, ip, statefull*. The value of *sndq* is a list of outgoing messages of the form $(!msg^*)$, representing a processed message to be returned to the corresponding *mc* that requested the service while the value of *rcvq* is a list of incoming messages of the form *(!msg,!sndr,amap)* representing the arrival of message *msg* to be processed by the protocol. The values of *pr*, *rst* and *ip* are meta representations of the protocol prerequisites, restrictions and interaction parameters respectively.

```
<cp:CPC|sndq:omsg,rcvq:imsg,pr:prlst,rst:rstlst,ip:iplst>
```

Messenger: A meta object of class *Msgr* has attributes *up, dwn, lastmsgid, lastmsl, msgclass*. The values of *up* and *dwn* are lists of incoming and outgoing messages of the form *(!msg,!sndr,amap)*, representing raw messages. The values of *lastmsgid* and *lastmsl* are meta representations of the message identification and message service list respectively.

```
<msgr:MsgrC|up:romsg,dwn:rimsg,lastmsgid:lmsgid,
      lastmsl:lmsl>
```

We briefly describe how the model isolated the synchronization requirement and enabled an asynchronous non-blocking session refresh without a connection oriented channel. Every generated key is associated with a key identifier (keyid) and every message is tagged with the corresponding keyid of the key used to encrypt the message. In order to decrypt messages, a list of recently used keys and their corresponding keyids is used. Thus, every session refresh adds the current key to the list of older keys and the newly generated key is used as the current key. Since protocols are described by listing out the (ordered) sequence of messages that participants would execute and exchange in a run, protocols consist of at least two roles: the initiator role and the responder role. The following rule formalizes the last step in the session refresh.

```
<B:Responder|secKey:SKB,aId:A,stage:3,ecom:A,counter:Q,
   session:sid,ciphers:cipher,skey:Skey,keylist:nil,
   queue:M6(encrypt((NB,sid,cid,new-pre-master),SKey))>
=>
<B:Responder|secKey:SKB,aId:A,stage:4,ecom:A,counter:Q+1,
   session:sid,ciphers:cipher,skey:newSkey,
   keylist:(Skey,cid),queue:none>
if ((NB==nonce(B,Q)) and (sid==getSid(decrypt(M6,SKey))))
```

Upon receiving an encrypted message with the required information from the responder, the initiator creates a new connection and sends an encrypted message *M6* to the responder containing the session identifier *sid*, the new connection identifier *cid*, which serves as a key identifier or *keyid*. Upon *M6* reception and decryption, the responder stores its current key and corresponding keyid and updates its secret key.

2. EXPERIMENTAL EVALUATION

Although symbolic execution allows us to investigate certain scenarios that might be hard to generate in an experimental setting, quantitative information is needed to evaluate the overhead and feasibility of our approach. Therefore, we developed a Maude API to systematically translate Maude commands into Java and enhance the existing communication framework proposed in [4] to take commands from the Maude engine via the API. This gives us the opportunity to model, observe and evaluate protocol composition and interaction, and show an integrated approach where formal prototyping and a real system implementation complement each other. We determined the reflection overhead introduced in the communication framework by categorizing the different overheads involved in the messaging process. First, we measured the message transmission and reception overheads of the underlying network. We then measured the time needed to send and receive messages through the RCF using specific case studies.Finally, we measured the overhead of the Maude-API by sending and feeding information to the semantic model in Maude.

3. CONCLUDING REMARKS

The focus of the work presented in this paper is on developing a semantic model based on rewriting logic with which to specify and reason about protocol interactions. The significance of our model is that protocols perform local state transitions, which can be naturally represented as rewrite rules. This allows us to use axiomatic specification at a reasonably high-level while still maintaining executability, which is used to identify critical properties, facilitate reasoning about them and speed up the lengthy proof process. We envision several directions for future work. One direction is to integrate the concept of time, so timely delivery and real-time constraints can be modeled. A second direction is to develop lightweight techniques that allow us to better reason about interactions between protocols as well as protocols and services.

4. ACKNOWLEDGMENTS

This research was supported by the Office of Naval Research under MURI Research Contract N00014-02-1-0715.

5. REFERENCES

[1] J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. In *Theoretical Computer Science 96(1):73-155*, 1992.

[2] J. Meseguer and C. Talcott. Semantic Models for Distributed Object Reflection. In *Proceeding of the European Conference on Object-Oriented Programming*, 2002.

[3] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and C. Talcott. The Maude 2.0 System. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications (RTA 2003)*, Lecture Notes in Computer Science, pages 76–87. Springer-Verlag, June 2003.

[4] S. Gutierrez-Nolasco and N. Venkatasubramanian. A Reflective Middleware Framework for Communication in Dynamic Environments. In *Proceedings of the International Symposium on Distributed Objects and Applications*, 2002.