Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 203 (2008) 95–106
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

A Formal Framework for Interactive Agents

Carolyn L. Talcott[1,2]

Computer Science Laboratory
SRI International
Menlo Park, CA 94025, USA

Abstract

This paper proposes a formal framework and architecture for specification and analysis of interactive agents. The framework can be used to explore the design space, study features of different points in the design space, and to develop executable specifications of specific agents and study their interactions with the environment. A long term goal is development of reasoning principles specialized to different regions of the design space.

Keywords: interaction, coordination, distributed object reflection, policy, autonomy

1 Introduction

The question of interaction versus algorithms as models of computation was raised by Wegner in [19]. Since then there has been much discussion of both philosophical and mathematical distinctions (c.f. [20,10,11]).

We are interested in what new issues arise, and how to take advantage of the fact that interaction goes beyond turing computability in designing interactive agents. For example, interactive agents can be concerned with issues such as survivability and situation awareness that are not relevant to an algorithm. An interactive agent may not only be aware of its surroundings, it may also affect its environment. It may need to negotiate, cooperate, or compete. We propose the following features to consider in the design of interactive agents.

- An agent has a boundary consisting of points of interaction with the environment. From the outside only what crosses the boundary is visible. Interaction points could be sensors, such as light detectors or thermometers, effectors such as switches or dials, or message queues for exchange of messages with other agents.

[1] The work was partially supported by NSF grant CCR-023446.

[2] Email: clt@cs.stanford.edu

doi:10.1016/j.entcs.2008.04.088

- An agent has actions that it can execute. It may also have goals, knowledge (about its environment and itself), policies constraining actions, or strategies for achieving goals.
- Internally an agent may have multiple concurrent activities; observing and processing sensory information; refining goals to subgoals, choosing actions, executing actions; evaluating and analyzing results: did actions have expected effect? updating knowledge by learning and inference;
- Interactivity means internal processes must be interruptible.

This richness of concerns and features leads to the questions of principles for designing interactive agents. In this paper we present initial ideas for an architecture and formal framework for design, specification, and analysis of interactive agents. The framework is based on rewriting logic and a reflective model of coordination for managing an agents activities. New forms of interaction are introduced to model both message and channel/signal based interactions, and to pave the way for modeling continuous interactions. The compositional interaction semantics of [18,7] is extended. The aims of the framework include:

- a higher level means of specifying and understanding agent behavior
- a place to classify agents with different 'skills'
- a formal design space to represent a variety of design decisions and to study trade-offs resulting from decisions such as adaptability vs. predictability;

One advantage of the proposed framework are that specifications are executable, allowing prototyping of designs at many stages. In addition, they are formally analyzable using the Maude rewriting logic system, and connections with other formal systems.

Section 2 is a brief overview of motivations and formal foundations for the framework. Section 3 describes the formal architecture and section 4 briefly describes the interaction semantics. In section 5 we analyze two agent systems using the formal framework. Section 6 summarizes and discusses future directions.

2 Background

To provide some context we give an overview of the autonomous agent systems whose analysis formed a starting point for the current work. This is followed by a brief introduction to the underlying formalisms: rewriting logic and the Reflective Russian Dolls (RRD) model of distributed object reflection.

2.1 Autonomous Agent Systems

The Policy And GOal based Distributed Architecture (PAGODA) (see `http://pagoda.csl.sri.com`) is a framework for specifying systems of autonomous agents. The

PAGODA architecture was inspired by the study of architectures developed for autonomous space systems, especially the Mission Data Systems (MDS) architecture

[9] and its precursors. Two essential features of MDS are goal-oriented operation and state variables that hold all the system knowledge. An executable specification of a simple Robot on a Grid (GridBot) based on the MDS architecture is described in [8].

A PAGODA system is a collection of PAGODA nodes under the control of a distributed node coordinator (DC). A PAGODA node is a collection of components coordinated by a node coordinator (NC). Both coordinators are policy-based. The job of the NC is to make sure components only get the messages they need and expect, and in the order expected. It also takes care of event notification, and logging certain events for self-evaluation and diagnostics. The job of a DC is to control dissemination of knowledge collected locally, deciding what to share, when, with whom. Coordination in PAGODA is discussed in more detail in [17]. A PAGODA reasoner based on soft constraint solving is described in [21].

2.2 Rewriting Logic

Rewriting logic [13] is a logical formalism designed for modeling and reasoning about concurrent and distributed systems. It is based on two simple ideas: states of a system are represented as elements of an algebraic data type; and the behavior of a system is given by local transitions between states described by *rewrite rules*. A rewrite rule has the form $t \Rightarrow t'$ if c where t and t' are terms representing a local part of the system state, and c is a condition on the variables of t. This rule says that when the system has a subcomponent matching t, such that c holds, that subcomponent can evolve to t', possibly concurrently with changes described by rules matching other parts of the system state. The process of application of rewrite rules generates computations (also thought of as deductions). Maude (`http://maude.cs.uiuc.edu`) is a system based on rewriting logic used for developing, prototyping, and analyzing formal specifications.

2.3 Reflective Russian Dolls

Reflective Russian Dolls (RRD) [14] is a formal model of distributed object reflection based on rewriting logic. The model combines logical reflection with a structuring of distributed objects as nested configurations of meta-objects (a la Russian Dolls) that can reason about and control their sub-objects. This model can be used to develop formal specifications of interaction as well as architectural, and behavioral aspects of distributed object-based systems. In this formalism, a *coordinator* is an object with a distinguished attribute that holds a nested configuration of objects and messages. The nested configuration could itself consist of base-level objects or coordinators each with their configuration of coordinated objects. The rewrite rules for a coordinator object control delivery of messages in its contained configuration. In [16] a special form of RRD called policy based coordination (PBRD) was introduced. Here each coordinator has three additional distinguished attributes: a policy, a policy state, and a queue of messages pending delivery to the nested configuration. In this case coordinator rewrite rules interpret the policy attributes, selecting a message to process and specifying what to do with it.

3 Framework for Interactive Agents

The proposed formal representation of interactive agents uses the PBRD coordination model as a starting point. PBRD is extended by giving interaction labels to rules specifying basic interaction steps; refining the notion of object interface; and specifying rules for interaction between an agent and its environment, including propagation of environment interactions through the nested object hierarchy. A compositional interaction semantics is derived from the executable semantics of interactive agents.

An alternating display, adapted from [1], is used as a running example to illustrate the structure and rules for interactive agents. This agent has three interaction points, two from which it reads sensors (say time and temperature) and one on which it writes a bitmap for display. The coordinator ensures that the display alternates between the two sensors. This example illustrates basic object behavior and use of sensors and effectors as well as message passing. We also use it to illustrate interaction patterns and different forms of composition.

3.1 Structure of Interactive Agents

We work in the context of a module IA that declares language and syntax for RRD objects, as well as sorts and operations for policies and interaction points. The following is a brief summary. Interactive Agents are formalized as PBRD objects having the form

```
[a : A | {C}, policy: P, policyState: pS, pending: pQ, atts | ips]
```

where `a` is the agents object identifier and `A` is it class identifier, a subclass of `InteractiveAgent`, and `ips` is the agents set of interaction points. Between the vertical bars are the agents attributes: `{C}` is the set of activities coordinated by `a`—a configuration (set) of interactive objects; `P` is the agents coordination policy; `pS` is its policy state; `pQ` is a queue of messages pending delivery; `atts` is a set of additional agent specific attributes. The nesting terminates with *base-level objects*, interactive objects whose configuration attribute is empty. For base-level objects the `policy`, `policyState`, and `pending` attributes are not used and can be omitted.

3.2 Rules of interaction

We consider four types of interaction point: `i(id,mQ)` (message input), `o(id,mQ)` (message output), `r(id,v)` (read a value), and `w(id,v)` (write a value). Here `id` is the name or identifier, `mQ` is a message queue, and `v` is a value. This notation is also used to denote interactions that label transitions and form the basis of the interaction semantics.

Rules for basic agent interactions.

A basic interaction rule describes response to a message `msg` in an input queue. There are two flavors—one silent, and one that writes a new value to a write interaction point. In both cases the message is removed from the input queue. A silent rule has the form

```
    [a : A | atts | i(id,msg mQ), ips]
    =[]=>
    [a : A | atts | i(id,mQ), ips1]
```

and a rule that writes has the form

```
    [a : A |  atts | i(id,msg mQ), w(ix,u), ips]
    =[w(ix,v)]=>
    [a : A |  atts | i(id,mQ), w(ix,v), ips1]
```

where in both cases `ips1` differs from `ips` at most by adding messages to queues of output interaction points. Between the `[]`s is the transition's interaction label where `[]` is the silent interaction.

Example: Reader and Writer agents.

The activities of an alternating display are instances of sensor reader and display writer objects. A reader, `or`, knows the name of its sensor, `sid`, and to whom it should report, `od`. When `or` receives a tick message `or<-tick` it sends the sensor reading, and sends itself another tick. This is expressed by the following rule.

```
[or : Reader | display : od, sensor: sid, atts
             | i(in,or<-tick iQ), o(out,oQ), r(sid,t)]
=[]=>
[or : Reader | display : od, sensor: sid, atts
    | i(in,iQ), o(out,oQ od<-sensed(sid,t) or<-tick), r(sid,t)]
```

where `iQ` and `oQ` are message queues. A writer simply computes a bitmap and writes it to the display. The previously written bitmap is recorded in `w(disp,obmap)`.

```
[od : DispA | | i(in,od<-sensed(sid,t) iQ), w(disp,obmap)]
=[w(disp,bmap)]=>
[od : DispA | | i(in,iQ), w(disp,bmap)]
if bmap :=  mkBmap(sid,t)
```

Coordination rules.

Coordination policies are specified by axioms for a function `next` that determines the next coordination actions. The axioms are of the form

```
   next(P,pS,pQ) = {dnQ, outQ, pS1, pQ1}  if cond
```

where `P` is a policy, `pS` is a policy state, and `pQ` is a queue of interactions pending processing. These interactions are of the form `u(o,ix,msg)` (up from the output `o(ix,-)` of object `o`) or `i(ix,msg)` (a message received in the coordinators input queue `i(ix,-)`). On the right, `dnQ` and `outQ` are lists of delivery actions of the form `(o,ix,mQ)`. Those in `dnQ` are for delivery to nested objects and those in `outQ` are for delivery to external objects. The rule for policy interpretation is the following.

```
[a : A | {C}, policy: P, policyState: pS, pending: pQ, atts
       | ips] =[]>
[a : A | {C1}, policy: P, policyState: pS1, pending: pQ1, atts
       | ips1]
if {dnQ, outQ, pS1, pQ1} := next(P,pS,pQ)
/\ C1 := deliver(C,dnQ)
/\ ips1 := emit(ips,outQ)
```

where `pS1` and `pQ1` are updated policy state and pending interaction queues, respectively. For `(o,ix,mQ)` in `dnQ`, `deliver(C,dnQ)` adds `mQ` to the input interaction point with identifier `ix` of the object with identifier `o`, while messages in `outQ` are added to output interaction points by `emit(ips,outQ)`.

There is a coordinator rule that moves messages from output interaction points in the nested configuration to the pending queue, and one that moves messages in its receptionist queues (input queues with identifier that of a nested object input queue visible to the environment) to the pending queue.

Alternating display coordinator policy.

The policy `altP` for the alternating display agent has a policy state of the form `(od,oQ)` where `od` is the name of the display object and `oQ` is a queue of reader object ids (thus it will work for alternation of any number of sensor inputs). Alternation of messages to the display is expressed by

```
next(altP,(od,(o1 o2 ...)), pQ0 u(o1,out,od<-sensed(s,t)) pQ1)
 =
{(od,in,od<-sensed(s,t)),nil,(od,(o2 ... o1)), pQ0 pQ1}
if pQ0 contains no elements of the form u(o1,out,od<-sensed(s1,t1))
```

while tick messages to enable reading are delivered as soon as they reach the front of the pending queue.

```
next(altP,(od,(o1 o2 ...)), u(o,out,o<-tick) pQ1)
 =
{(o,in,o<-tick,nil,(od,(o2 ... o1)), pQ1}
```

An alternating display agent does not exchange messages with any other agents. It only reads time and temperature sensors and writes to the display. Thus it has three interaction points: `r(time,-)`, `r(temp,-)`, `w(disp,-)` (we use - to indicate unspecified value). The initial configuration of an alternating display agent looks like the following

```
[a : AltDisplay |
    { [o1 : Reader | display: od, sensor: time,
                   | i(in,o1<-tick), o(out,nil), r(time,t0)]
      [o2 : Reader | display: od, sensor: temp,
                   | i(in,o2<-tick), o(out,nil), r(temp,t1)]
      [od : DispA | | i(in,nil), w(disp,blank)]  },
   policy: altP, policyState: (od,o1 o2), pending: nil
  | r(time,t0), r(temp,t1), w(disp,blank)]
```

According to the coordination and behavior rules, `o1` and `o2` will output `sensed` messages. The message from `o1` will be delivered to `od` first, then the message from `o2`. Tick messages will be delivered to `o1` (`o2`) after their `sensed` messages are delivered, and the process repeats.

Rules for interaction with environment.

The underlying rewriting logic semantics says that silent transitions of a nested configuration lift to transitions of the whole agent. Messages from external agents are placed in the queues of input interaction points where they can be taken from the queue by the agent.

```
[a : A | { C }, atts | ips, i(ix,iQ)]
 =[i(ix,msg)]=>
[a : A | { C }, atts | ips, i(ix,iQ msg)]
```

Dually messages to external agents are placed in the queues of output interaction points, and removed by the environment (not shown).

Read/write interaction points are for sensing and effecting the agents exterior. An agent may silently read values from read interaction points. The value is only changed by the environment. This change is seen by all nested agents with this interaction point.

```
[a : A | { C }, atts | ips, r(ix,v)]
= [r(ix,u)] =>
[a : A | { pushRead(C,ix,u) }, atts | ips, r(ix,u)]
```

An agent may write new values into write interaction points, to affect the environment. A write interaction is propagated to the containing agent (formalized using conditional rewriting).

```
 C   =[w(ix,v)]=> C1
---------------------------
[a : A | { C }, atts | ips, w(ix,u)]
 =[w(ix,v)]=>
[a : A | { C1 }, atts | ips, w(ix,v)]
```

4 Interaction Semantics

An *interaction path* is a (possibly infinite) sequence of interactions. Each computation of an agent (allowed by the rewrite rules) gives rise to an interaction path consisting of the sequence of (non-silent) interactions labeling the transitions (rewrite rule applications). The semantics of an interactive agent is thus the set of interaction paths of its possible computations. This definition derives from earlier work developing interaction semantics for actors [18,7] ideas from Timed Data

Stream semantics for the Reo coordination model [2] and signal event semantics [12]. Interaction semantics is similar in spirit to the Interactive Stream Languages of [10]. The ideas are also related to work on *interfaces* of reactive and concurrent systems such as, for example, [5,6]. As an example, a possible interaction path of the alternating display is

```
   r(time,600)
   r(time,700)
   r(temp,20)
   w(disp,mkBmap(time,600))
   w(disp,mkBmap(temp,20))
   r(temp,21)
   r(time,800)
   w(disp,mkBmap(time,800))
   w(disp,mkBmap(temp,21))
```

Since reads are controlled by the environment, there may be reads that are not observed by the `Reader` and thus not reflected in the display sequence, for example `700` is not displayed.

Interaction semantics is compositional both vertically and horizontally. The semantics of the horizontal (parallel) composition of two systems is done by zipping compatible paths one from the semantics of each system. Two paths are compatible if their subsequences of complimentary interactions, such as out/write in one and in/read in the other match. In the composed path, these interactions interactions become silent transitions and disappear, and the remaining interactions are merged. (See [18] for details in the case of horizontal, actor-actor composition, and [7] for vertical, actor-metaactor, composition.) For example, consider the semantics of a system `s1` consisting of the two readers, and another `s2` consisting of the display writer. A compatible pair of interaction paths for these systems is

```
S1:                         S2:
   r(time,600)
   r(time,700)
   r(temp,20)
   o(od,sensed(time,600))   i(od,sensed(time,600))
   r(time,800)
                            w(disp,mkBmap(time,600))
   o(od,sensed(time,700)    i(od,time(700))
                            w(disp,mkBmap(time,700))
   r(temp,21)
   o(od,sensed(temp,20))   i(od,sensed(temp,20))
   ...
                             w(disp,mkBmap(temp,20))
                          ...
```

Composing this pair results in an interaction path for the parallel composition `s1 s2`.

```
r(time,600)
r(time,700)
r(temp,20)
w(disp,mkBmap(time,600))
r(time,800)
w(disp,mkBmap(time,700))
r(temp,21)
w(disp,mkBmap(temp,20))
....
```

Notice that this path does not satisfy the alternation requirement. Such is life.

To treat a coordinator (meta-object) as a separate component independent of its nested configuration, conceptually, we replace its configuration attribute by interaction points, one for each messaging interface of a contained object. For this purpose we introduce up and down interactions. Up interactions of a coordinator have the form `u(o,ix,msg)` and synchronize with interactions `o(ix,msg)` of an object with identifier `o`. (These appear in the pending interaction queue in the composed system.) Down interactions of a coordinator have the form `d(o,ix,msg)` and synchronize with interactions `i(ix,msg)` of an object with identifier `o`.

The requirements for an alternating display coordinator can be expressed as a relation Alt_{IO} on the interactions of the coordinatees (similar to Abstract Behavior Type specifications of sets of Timed Data Stream [1]) as follows

$$\begin{aligned}\theta \in Alt_{IO} \;\Leftrightarrow\; & \pi(\theta, d(od, in, -))(2i) = \pi(\theta, u(ot_1, out, -))(i) \wedge \\ & \pi(\theta, d(od, in, -))(2i+1) = \pi(\theta, u(ot_2, out-))(i) \wedge \\ & ix(\theta, u(ot_1, out, -), i) < ix(\theta, d(od, in, -), 2i) \wedge \\ & ix(\theta, u(ot_2, out, -), i) < ix(\theta, d(od, in, -), 2i+1)\end{aligned}$$

where $\pi(\theta, d(od, in, -))$ is the subsequence of messages of interactions $d(od, in, msg)$ occurring in θ, and $ix(\theta, d(od, in, -), i)$ is the index in θ of the ith element of $\pi(\theta, d(od,\, in, -))$. Similarly for $\pi(\theta, u(o_j, in, -))$ and $ix(\theta, u(ot_j, out, -), i)$.

The claim is that composing a coordinator satisfying Alt_{IO} with a system such as S1 S2 will result in an alternating display. In this case composition is placing the system into the coordinators configuration attribute.

5 Goal-based autonomous agents

As a small step towards using the interactive agent framework to analyze agent structure we consider the kinds of activities that might make up a goal-based autonomous agent and related design points. We use these to analyze two autonomous agent systems. The activities we consider are: knowledge manager (KM), estimator, controller, goal achiever, analyzer, and monitor.

A *KM* encapsulates knowledge: models, policies, history, situation/context It accepts requests to modify the knowledge base and queries to retrieve information from the knowledge base.

An *estimator* deals with reading sensors and converting what is read into higher level data. It could just be an identity function, but provides a means for interacting with lower level sensors, for example by converting a voltage reading to a temperature, pressure, or liquid level. An estimator may proactively read sensor information and tell the KM its interpretation, or it may only read upon request. A controller converts basic action requests to effects on the environment, setting knobs or switches, turning motors on or off, etc..

A *goal achiever* accepts goal requests (constraints to be satisfied, something to be achieved) and produces responses which contain basic actions to perform and assumptions made in choosing these actions. A goal achiever may (should) use knowledge from the KB to decide on a course of action. It may reach decisions by some form of reasoning (planning, constraint solving, search), or by a simple table lookup.

An *analyzer* derives new knowledge from information already in the knowledge base, or corrects existing knowledge. It can request 'experiments to be done' to gather more information. A learner is an example of an analyzer. The learner's objective is typically to fill in or correct model parameters. A trust manager is another kind of analyzer. It observes interactions with other agents and builds trust models, (X can be trusted for Y with confidence L) that might be used by a goal achiever, or other analyzers. Yet another kind of analyzer, which we call a *thinker*, builds new models, or refines existing models by drawing inferences from existing knowledge and observations.

A *monitor*'s job is checking assumptions that are expressed as constraints on observables: functions of sensor readings and messages received. It may make periodic checks, or only upon specific request. It may always report (for logging in KB) the results of a check, or only report when a check fails. Typically the assumptions are generated by a goal achiever, but an analyzer could also make hypotheses to be checked.

Using the above notions we can characterize the two autonomous agent systems mentioned in Section 2: the MDS based GridBot; and the PAGODA software defined radio (SDR).

The GridBot has a single goal achiever (called a goal elaborator). Its remaining activities are organized by state variable. For each state variable there is a knowledge manager (the state variable itself), a sensor, an estimator, and a controller. The simple GridBot has just three state variables: its motion base (for moving and turning); a camera, and a battery. Coordination is done by a scheduler. In general MDS schedulers have a policy that specifies how frequently to visit each state variable, and for each visit a fixed order of executing steps of the associated activities.

The PAPGODA SDR has a single knowledge base that stores goals, situation information (for example, mission phase and status), environment information, a model of how knob settings effect the radio's performance, and a history of actions and sensor readings. All sensor/effector interaction points are encapsulated in the HAL (Hardware Abstraction Layer). HAL handles all requests for actions (knob settings) and sensor reading. There is one reasoner (for which two implementations exist) that accepts goals and sends knob settings to the HAL and monitoring tasks to the monitor. There is a learner that can be activated to learn initial model parameters or correct existing parameters. The PAGODA coordination policy ensures that goal requests, knob settings, sensor readings and monitor reports are logged in the knowledge base and that the learner is notified of events of interest to it. It also ensures that goal requests to the monitor are serialized. Otherwise activities go on concurrently.

6 Conclusion and Future Work.

We have described an architecture and formal framework for specifying and analyzing interactive agents. The main ideas are policy-based coordination of multiple agent activities and the notion of interaction points. The compositional semantics of actors and reflective objects is extended to the richer interaction mechanisms.

Broy's characterization of components as functions that transform data streams [4,3] is similar in spirit to interaction path semantics. The former focuses on data flow while the latter on interaction events and supports modeling dynamic interconnections and richer composition mechanisms. The Abstract Behavior Type semantics of Reo using Timed Data Streams (TDS) [2,1] is much closer to our enriched interaction semantics, although Reo is concerned with channel-based communication while our approach is concerned with messages and signals. In [15] a mapping between interaction paths and TDS is sketched for the special case of buffered channels in Reo.

There are several interesting directions for future work. The extension of actor interaction semantics drew on ideas from Timed Data Streams [2] and work on signal semantics and causal interfaces [12]. An in depth comparison of these and other semantic models for interaction is needed.

There are a number of variations on the details of the interaction semantics described here that should be understood. For example single interactions could be replaced by sets of concurrent interactions where order is undetermined; considering continuous signals driving read/write interactions rather than discrete changes; and adding a notion of time.

Last but not least is developing logical rules and principles for inferring emerging behavior of interactive agents. This goes hand in hand with developing design principles. One of the motivations for policy based coordination is to set the stage for compositional reasoning, that is being able to use constraints enforced by coordination policies to be able to simplify reasoning about the coordinated activities.

References

[1] F. Arbab. A behavioral model for composition of software components. *L'Objet*, 12:33–76, 2006.

[2] F. Arbab and J.J.M.M Rutten. A coinductive calculus of component connectors. In *WADT'02*, volume 2755 of *LNCS*, pages 34–55, 2002.

[3] M. Broy and G. Stefanescu. The algebra of stream processing functions. *Theoretical Computer Science*, 258, 2001.

[4] M. Broy and K. Stolen. *Specification and development of interactive systems*, volume 62 of *Monographs in Computer Science*. Springer-Verlag, 2001.

[5] L. de Alfaro and T. A. Henzinger. Interface automata. In *Ninth Annual Symposium on Foundations of Software Engineering (FSE)*, pages 109–120. ACM Press, 2001.

[6] L. de Alfaro and T. A. Henzinger. Interface theories for component-based design. In *1st Intl. Workshop on Embedded Software*, volume 2211 of *LNCS*. Springer-Verlag, 2001.

[7] G. Denker, J. Meseguer, and C. L. Talcott. Rewriting semantics of distributed meta objects and composable communication services. In *Third International Workshop on Rewriting Logic and Its Applications (WRLA'2000)*, volume 36 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2000.

[8] G. Denker and C. L. Talcott. Formal checklists for remote agent dependability. In *Fifth International Workshop on Rewriting Logic and Its Applications (WRLA'2004)*, volume 117 of *ENTCS*. Elsevier, 2004.

[9] D. Dvorak, R. Rasmussen, G. Reeves, and A. Sacks. Software Architecture Themes In JPL's Mission Data System. In *IEEE Aerospace Conference, USA*, 2000.

[10] D. Goldin, S. Smolka, P. Attie, and E. Sonderegger. Turing machines, transition systems, and interaction. *Information and Computation Journal*, 194(2):101–128, 2004.

[11] D. Goldin and M. Viroli, editors. *Foundations of Interactive Computation (FInCo 2005)*, volume 141 of *ENTCS*. Elsevier, 2005.

[12] E. A. Lee. Concurrent semantics without the notions of state or state transitions. In *Formal Modeling and Analysis of Timed Systems*, LNCS, pages 18–31. Springer, 2006.

[13] J. Meseguer. Conditional Rewriting Logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[14] J. Meseguer and C. L. Talcott. Semantic models for distributed object reflection. In *European Conference on Object-Oriented Programming, ECOOP'2002*, volume 2374 of *LNCS*, pages 1–36, 2002.

[15] S. Ren, M. Sirjani, and C. Talcott. Comparing three coordination models: Reo, arc, and rrd, 2007. in preparation.

[16] C. Talcott. Coordination models based on a formal model of distributed object reflection. In *1st International Workshop on Methods and Tools for Coordinating Concurrent, Distributed and Mobile Systems (MTCoord 2005)*, 2005.

[17] C. Talcott. Policy-based coordination in pagoda: A case study. In *2nd International Workshop on Methods and Tools for Coordinating Concurrent, Distributed and Mobile Systems (MTCoord 2005)*, 2006.

[18] C. L. Talcott. Composable semantic models for actor theories. *Higher-Order and Symbolic Computation*, 11(3):281–343, 1998.

[19] P. Wegner. Why interaction is more powerful than algorithms. *CACM*, May 1997.

[20] P. Wegner and D. Goldin. Computation beyond turing machines. *CACM*, April 2003.

[21] M. Wirsing, G. Denker, C. Talcott, A. Poggio, and L. Briesemeister. A rewriting logic framework for soft constraints. In *Sixth International Workshop on Rewriting Logic and Its Applications (WRLA'2006)*, ENTCS. Elsevier, 2006.