Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 181 (2007) 97–112
www.elsevier.com/locate/entcs

Policy-based Coordination in PAGODA: A Case Study

Carolyn L. Talcott[1,2]

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

Abstract

PAGODA (Policy And GOal Based Distributed Autonomy) is a modular architecture for specifying and prototyping autonomous systems. A PAGODA node (agent) interacts with its environment by sensing and affecting, driven by goals to achieve and constrained by policies. A PAGODA system is a collection of PAGODA nodes cooperating to achieve some mutual goal.
This paper describes a specification of PAGODA using the Russian Dolls model of policy-based coordination. In PAGODA there are two forms of coordination: local and global. Local coordination is used to compose the components of a PAGODA node. The local coordinator is concerned with ensuring component level synchronization constraints, cross component message ordering constraints, routing of notifications, and interaction with the external world. The global coordinator is concerned with dissemination of information, negotiation of responsibilities, and synchronization of activities. Requirements for a PAGODA node coordinator are given and an example set of policies is specified. Principles for showing that the policies satisfy the requirements are discussed as a first step toward a logic of policy-based coordination. Development of a distributed coordinator is the subject of ongoing work. Some challenges and possible solutions are discussed.

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

1 Introduction

There is a growing interest in autonomous agents that interact with and affect their environment, and have some ability to observe, reason, and adapt. As part of a larger system agents should also be able to compete for resources but also to cooperate for mutual benefit or to achieve an overall goal.

PAGODA (Policy And GOal based Distributed Autonomy) is a modular architecture for design of interactive autonomous systems. A PAGODA system is a collection of PAGODA nodes cooperating to achieve some mutual goal. A PAGODA node (agent) interacts with its environment by sensing and affecting, driven by goals

[1] The work was partially supported by NSF grant CCR-023446.
[2] Email: clt@cs.stanford.edu

doi:10.1016/j.entcs.2007.01.056

to achieve and constrained by policies. The PAGODA architecture was inspired by studying architectures developed for autonomous space systems, especially the MDS architecture [15] and its precursors [25]. Policy based coordination is used at two levels: local modular combination of components making up an agents behavior; and coordination of a distributed system of agents constraining the possible interaction scenarios to meet end-to-end requirements. PAGODA has been developed in the context of projects providing driving applications, including a rover (for example for exploration or patrol) [13,14] and software defined radios [35] supporting specific missions. Other potential applications include reactive/adaptive planners, cognitive radios, software assistants, and self-configuring systems.

The long term objective of the PAGODA project is to develop techniques for specification and analysis that take advantage of the modularity and the declarative nature of policy- and goal-based systems. Our approach is based on the Reflective Russian Dolls (RRD) model of distributed object reflection [23,34] which in turn is founded on the rewriting logic formal modeling framework [22,24]. In [34] a general approach to modeling policy-based coordination using RRD was presented. The contribution of the present paper is presentation of a substantial case study illustrating the benefits of policy-based coordination for modularity of design and reasoning.

2 Background

To provide some context we give a brief introduction to rewriting logic and the Reflective Russian Dolls (RRD) model of distributed object reflection. *Rewriting logic* [22,24] 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). Rewriting logic is reflective (capable of faithfully representing important aspects of its own syntax and deductive/computation mechanisms, see [10]). Maude [8,9] is a formal language and tool set based on rewriting logic used for developing, prototyping, and analyzing formal specifications.

Reflective Russian Dolls (RRD) [23] is a general 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. At a high-level of abstrac-

tion, we model a coordinator as an object with a distinguished attribute that holds a nested configuration of objects and messages, a policy attribute, and additional attributes to represent processing state. The rewrite rules for a coordinator object control delivery of messages in its contained configuration.

The nested object coordination model provides a relatively simple formal executable model for specifying and analyzing system interaction properties. As discussed in [34] such models can be systematically transformed into flat object system specifications that exhibit equivalent behavior, thus providing a principled path to implementation. Flattened coordinator based systems can also be executed for prototyping and lower-level analyses.

3 PAGODA

The picture in Figure 1 gives an abstract view of a goal driven autonomous system that interacts with its environment.

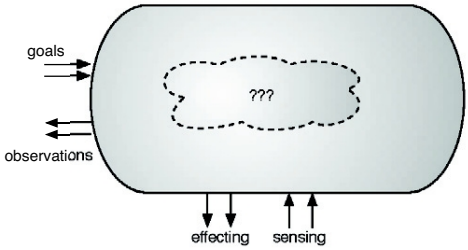

Fig. 1. Goal driven autonomous system

This picture could be interpreted as representing a single agent, or collection of agents. The difference appears when considering the internal structure, and the degree to which a notion of global/system-wide state is meaningful.

The question addressed by PAGODA is how to specify autonomous behavior that meets or achieves its goals (subject to constraints on external conditions) in a modular and declarative manner using models of its environment. Our solution is to factor the behavior into components each with a specific role that combine to achieve the desired result.

3.1 PAGODA nodes

Figure 2 shows the principal components of a PAGODA node: a knowledge base (KB), a reasoner (R), a monitor (M), a learner (L), and a 'hardware' abstraction layer (HAL). These interact with each other and the environment under the control of a coordinator (C).

The knowledgebase (KB) is the centerpiece. It contains knowledge that is shared and updated by the remaining components. This knowledge inlcudes a wide range of information:

- *Goals* that specify what the node or system is trying to achieve. A goal could be

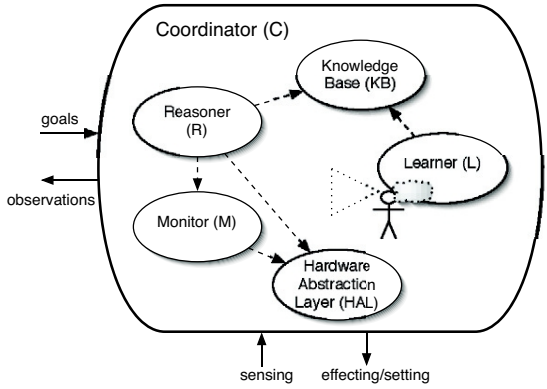

Fig. 2. PAGODA node architecture

a very high-level goal such as carrying out a scientific experiment or tuning parameters to achieve a given quality of service; or lower level goals that correspond to actions that can be carried out.

- *Policies* that constrain the allowed actions / interactions of a node or system. A policy might reduce the number of choices for setting parameters, for example based on importance of different competing effects. Another policy might determine trade-offs between speed and power usage. Other policies might control aggregation and abstraction of information used locally or communicated to other agents.
- A *device model* that specifies the HAL interface: parameters/knob that can be set (effecting) and read (sensing) and their relationships. At the system level the model should also specify how values sensed at different nodes can be combined to determine non-local system properties, and the relationships of such properties to higher level goals.
- An *environment model*, representing relevant features of the environment in which the node is operating, including information about other nodes. For a mobile node this could include terrain information or building maps.
- *Node state*, which includes values of variables determined by sensor readings and deduced from actions and information collected from other nodes. It also includes 'situation' information such as the stage in a complex task/mission or progress towards achieving a goal.
- *History*, a log of events–goals received, knob settings and sensor readings, monitor alerts, and so on.

The job of the *reasoner* component (R) is to determine proper parameter settings in response to goals requests: new goals, starting a new stage of a current goal, or alerts raised due to unexpected sensor values, indicating that adjustments need to be made. The reasoner uses information from the KB as a basis for its deductions: the device and environment model, the goals and policies, and the current state. When new parameter settings are determined, the reasoner also provides

justifications such as what sensor values and/or what relationships from the device model were used to infer the new settings. This can be used for diagnostics if things don't go as expected. The reasoner also specifies sensors that should be monitored and conditions on sensor reading under which the reasoner to be alerted to take corrective action.

The *monitor* component (M) receives monitoring tasks from the reasoner, reads and evaluates specified sensors, and sends alerts to the reasoner when sensor readings are not within specified limits.

The job of the *learner* component (L) is to improve the model used by the reasoner to infer appropriate knob settings. In passive mode it observes events such as goals, settings, sensor readings and alerts and attempts to improve relationships specified by the model based on this information. A learner may also have an active mode where it is allowed to propose experimental settings and observe the results.

The *hardware abstraction layer component* (HAL) is an interface to the sensors and effectors used by the node. It plays the role of device driver, handling knob setting and sensor reading requests. In a real system the HAL might map requests to a format that is understood by the actual hardware, or even to a lower level abstraction layer. The intent is that these interactions should obey the 'physics' specified by the device model, but the node needs to be prepared for things to go wrong—some hardware component breaks, the environment is different than expected, it is being operated outside the expected operational mode, and so on.

The coordinator (C) controls message semantics for internal components and mediates interactions with the external world. The coordinator is responsible for ensuring specified relationships between the events (message deliveries) seen by different components, and for meeting logging and notification requirements. It also enforces component level synchronization constraints (only delivering messages for which the component is enabled). The coordinator actions are specified declaratively by policies. Note that coordinator policies are similar in spirit, to policies used by the reasoner, but different in detail.

This architecture provides a simple means of plugging in different component instances. PAGODA node components interact with other node components based on component type not on component instance identity. Thus it is easy to have multiple reasoners, knowledge bases, learners, etc., by simply modifying the coordinator policy to choose appropriate component instances. Different reasoners might be appropriate for different situations or goals, knowledge might be split into categories and stored in separate KB instances, or two KB instances might contain knowledge at different levels of abstraction appropriate for different situations.

Additional components types could be easily incorporated. For example a component capable of knowledge abstraction or aggregation could be invoked from time to time by the coordinator to infer higher-level information from sensor data or information received from peers. Such a component could be used to raise the level of abstraction at which the reasoner or learner operates.

A formal executable specification of the PAGODA architecture has been developed in the Maude language [8,9] and instantiated with a very simple device model

of a radio to test the ideas. Goals are treated as soft constraints on subsets of sensor readings. The relationships between effectors (knobs) and sensor readings and between sensor readings and goals are formalized as constraint semi-rings, which provides a clean mathematical basis for solving soft-constraints [5,26,35]. In addition, initial experiments adding learner functionality have been carried out using both simple hill-climbing and a logic based learning algorithm. An additional component, MadRad, that simulates actual radio hardware/software including random, unusual and faulty behavior, and test scenarios have been developed to explore possible system behaviors by executing the composition of PAGODA and MadRad. The PAGODA specification and related documents are available form the PAGODA website http://pagoda.csl.sri.com.

4 Specifying a PAGODA Node Coordinator

4.1 Abstract policies

Rather than introduce a specific policy language, we treat policies as abstract entities whose meaning is axiomatized using functions that specify when a message is enabled for delivery, the messages to wait for, and the messages to deliver or put into the coordinators output queue. This is done by specifying data types (called sorts in Maude), operations to constructor elements of these data types, and functions to manipulate and test data. The latter are defined by equational axioms that specify how results of applying the functions can be computed.

Formally, we introduce a sort `Policy` and a set of 'policy interface functions'. We assume a sort `Msg` formalizing messages exchanged between objects, and introduce an auxiliary sort, `Wait4s`, defined to be sets of elements of the form `w4(?M,iMs,xMs)` where `?M` is a message pattern and `iMs`, `xMs` are sets of messages. Wait4s can be thought of as call-backs, or continuations. The message pattern specifies a message that is expected, (in response to some previously delivered message), and two message sets specify how the expected message is to be processed: messages in `iMs` are to be delivered to the internal configuration, and messages in `xMs` are to be sent to an object in some external node. Message patterns are constructed like messages with the addition of pattern variables (constants representing unspecified values) for each message component sort. Matching binds pattern variables to corresponding terms (thus determining the value represented by the pattern variable in in the context of the match). The boolean function `pending(w,x,MB)` returns true just if the wait4 set `w` contains `w4(msg(y,x,?MB),iMs,xMs)` where the pattern `?MB` matches `MB`.

The policy interface functions are the following (where `P` ranges over `Policy`, `msg` ranges over `Msg`, `w` ranges over `Wait4s`, and `q` ranges over `MsgQ` (lists of messages)).

- `deliverM(P,w,msg)` returns the set of messages to be delivered to the coordinated configuration.
- `updWait4s(P,w,msg)` returns a wait4 set obtained by removing any element of `w` that is matched by `msg` and adding any wait4 elements implied by processing of `msg`.
- `sendOutM(P,w,msg)` returns the set of messages to be sent out by the coordinator when

`msg` is processed, and

- If there is a message in `q` that is enabled according to policy `P` and wait4s `w`, then `firstEnabled(P,w,q)` returns a pair `{msg,q'}` where `msg` is the first such message and `q'` is the remaining message list. If there is no enabled message, an indication of failure is returned.

4.2 Coordinator objects and rules

A PAGODA node coordinator is formalized as a coordinator object in the RRD model. Using Maude like syntax a PAGODA node coordinator is an instance of

```
[c : C | {conf}, policy(P), que(q), wait4s(w) | cin, cout]
```

where `c` is the object identifier, `C` is the coordinator class identifier, and `cin`, `cout` are the coordinator's input and output message queues respectively. Between the vertical bars are the attributes representing the coordinator state. `policy(P)` represents the coordinators policy, and the `que(q)` and `wait4s(w)` attributes hold policy parameters. The attribute `q` is a list of messages to be processed, and `w` is of sort `Wait4s`. The configuration attribute `conf` consists of objects representing the components being coordinated. Each such object has a form similar to the coordinator, namely

```
[o :  O | oatts | oin, oout]
```

where `o` is the object identifier, `O` is the class identifier, `oatts` the object attributes, and `oin`, `oout` its message input and output queues.

The rewrite rule for policy based message processing is the following.

```
crl[process]:
 [c : C | {conf}, policy(P), que(q), wait4s(w) | cin, cout]
   =>
 [c : C
   | { deliver(conf,deliverM(P,w,msg) }, policy(P),
     que(q'), wait4s(updWait4s(P,w,msg))
   | cin, cout sendOutM(P,w,msg)]
 if {msg,q'} := firstEnabled(P,w,q) .
```

where `deliver(conf,msgs)` puts each message in the set `msgs` into the input queue of its target object. Given a coordinator object instantiating the rule premise (left-hand side), the rule can fire just if the rule condition can be satisfied, i.e just if there is an enabled message in `q` so that `firstEnabled` returns a message-queue pair rather than a failure indication. In this case the attributes of the coordinator object are updated according to the right-hand side of the rule.

There are two additional rules needed for the coordinator, one to move messages from its input queue to the internal processing queue, and one to move messages from output queues of objects in its configuration attribute to the internal processing queue. The three coordinator rules together with rules for each component make up the executable specification of a PAGODA node. We spare the reader from the details.

4.3 Example Policy Axioms

As an example, we describe the policy functions for the policy, `PP`, of the current PAGODA instance. Messages have the form `msg(to,from,MB)` where `to` is the message

target, `from` is the sender (or blank) and `MB` is the message body (sort `MsgBody`). We introduce the following sorts to classify PAGODA message bodies. For each sort, `Sort`, `?sort` is a variable ranging over `Sort` and `?Sort` is a pattern variable that matches any term of the corresponding sort.

- `Goal`, a goal to be achieved (includes alerts). `ack(?Goal)` is a pattern matching a goal acknowledgement message. A goal achiever should only send such a message when actions intended to achieve the goal have been taken.
- `Set*`, an annotated knob setting request (`actions(Set*)`) containing knob setting actions for HAL, monitoring task descriptions (`tasks(Set*)`), and possibly additional information that is not relevant for the coordinator.
- `Set`, knob setting actions. Elements of `Set` are elements of `Set*` with annotations omitted. `ack(?Set)` is a pattern matching a knob setting acknowledgment.
- `MonitorTask`, monitoring tasks
- `Sense`, `SenseReply` — sensor reading request / reply
- `Query`, `QueryReply` — knowledge base query / reply
- `UpDate`, knowledge base update request. `ack(?Update)` is a pattern matching an update acknowledgment.

The main requirements for the current PAGODA coordinator are enforcement of component level synchronization and message ordering constraints. The synchronization constraints specify when a component is enabled to receive a message.

- R is enabled for a `Goal` if it is not currently working on a goal. It is enabled for replies to `Sense` and `Query` messages that it sent and acknowledgments of its `Set` requests.
- KB is enabled for `Query`, `Update`, and `log` messages.
- HAL is enabled for `Sense` and `Set` messages.
- M is enabled for `MonitorTask` messages and for replies to `Sense` messages that it sent.
- L is enabled for `notice` messages and for replies to `Sense` messages that it sent.

Enforcement of these constraints means that a message is delivered to a component only if the component is enabled for that message.

The message ordering requirements for the coordinator are

- L sees `Update` and `log` messages in the same order as KB
- KB sees `Set` requests and `Sensor` replies in same order as HAL
- KB sees `Goal` messages, `Sensor` replies and `Set` requests in same order as R

The PAGODA policy `PP` is axiomatized by giving equations defining the values of the policy functions applied to `PP`, using the classification of messages, and guided by the informal requirements. We introduce two additional message body constructors: `log(msg)`— request to log a message, and `notice(msg)`—notification of message delivery. The following are an illustrative selection of these equations (in Maude-like syntax).

Policy for Goal messages. A goal message is enabled if there is no pending acknowledgement for some other goal (`?Goal`). The expected acknowledment is added to the wait4s. The expression `external(x)` is true if `x` is the identifier of an external node (goals may come from outside the node, or inside). The added wait4 specifies that the acknowledgment is sent out if the sender of the goal message is external, and delivered internally otherwise. The goal message is delivered to the reasoner, R, it is logged, and a notice is delivered to L.

```
enabled(PP,w,msg(R,x,?goal)) = not(pending(w,R,ack(?Goal))
updWait4s(PP,w,msg(R,x,?goal)) =
  (if external(x)
   then (w w4(msg(x,R,ack(?Goal)),
              none, msg(x,R,ack(?Goal))))
   else (w w4(msg(x,R,ack(?Goal)),
              msg(x,R,ack(?Goal)), none))
   fi)
sendOutM(P,w,msg(R,x,?goal)) = none
deliverM(P,w,msg(R,x,?goal)) =
     msg(R,x,?goal) msg(KB,-,log(msg(R,x,?goal)))
     msg(L,-,notice(msg(R,x,?goal)))
```

When a goal acknowledgement is processed, the binding to the pattern variable `?Goal` is used to instantiate the message to deliver, `msg(x,R,ack(?Goal))`.

Policy for Set* messages. An annotated set message is enabled if there are no pending set acknowledgments. The expected acknowledgment is added to the wait4s, specifying that the acknowledgment is to be delivered. The actions part is delivered to HAL, the monitoring tasks are delivered to M, the message is logged, and a notice delivered to L.

```
enabled(PP,w,msg(HAL,x,?set*)) =
  not(pending(w,HAL,ack(?Set)))
updWait4s(PP,w,msg(HAL,x,?set*)) =
  (w w4(msg(x,HAL,ack(?Set)), msg(x,HAL,ack(?Set)), none))
sendOutM(P,w,msg(HAL,x,?set*)) = none
deliverM(P,w,msg(HAL,x,?set*)) =
  msg(HAL,x,actions(?set*)) msg(M,x,task(?set*))
  msg(KB,-,log(msg(HAL,x,?set*)))
  msg(L,-,notice(msg(HAL,x,?set*)))
```

Policy for Sense messages. The axioms for sense messages are similar to those for set messages. A reply is expected, but the exact reply is not known, so a pattern is placed in the wait4 entry. When the expected message pattern is matched, occurrences of the pattern variable `?SenseReply` in the message delivery descriptions are instantiated.

```
enabled(PP,w,msg(HAL,x,?sense)) = true
updWait4s(PP,w,msg(HAL,x,?sense)) =
    (w w4(msg(x,HAL,?SenseReply),
       (msg(x,HAL,?SenseReply)
        msg(KB,-,log(msg(x,HAL,?SenseReply)))
        msg(L,-,notice(msg(x,HAL,?SenseReply)))),
       none))
sendOutM(P,w,msg(HAL,x,?sense)) = none
deliverM(P,w,msg(HAL,x,?sense)) = msg(HAL,x,?sense)
```

Using Wait4s. Some messages can only be processed if they are expected, namely replies and acknowledgments. A message `M` is expected if it matches a pattern `?M` of one of the wait4s. A successful match produces a substitution `s` for the pattern variables in `?M`. This substitution is used to instantiate the messages to be delivered (`iMs[S]`) and the messages to be sent out (`xMs[S]`).

```
enabled(PP,(w,w4(?M,iMs,xMs)),M) = M matches ?M
updWait4s(PP,(w,w4(?M,iMs,xMs)),M) = w
```

```
  sendOutM(PP,(w,w4(?M,iMs,xMs)),M) = xMs[S]
  deliverM(PP,(w,w4(?M,iMs,xMs)),M) = iMs[S]
 where S := match(M,?M)
```

Once a coordinator policy has been specified, and behaviors for the remaining PAGODA components specified (as maude rewrite rules) the system can be prototyped by defining a configuration containing an instance of each component together with a driver component, and a component simulating the external system being controlled. The role of driver component—called Headquarters (HQ) in the case of the software defined radio—is to send goals to be acheived. Using Maude's `rewrite` command, one possible execution of the system can be found. Using search, one can look for reachable system states that satisfy properties of interest, either good or bad states. As discussed below, one can also reason about the coordinator behavior independently of the components its coordinating.

4.4 Reasoning about Policy-Based Coordination

There are two reasoning/verification tasks associated with a coordinator policy. One is to show that the policy meets the coordination requirements, the other is to use the assumption that the coordination requirements are met to verify properties of the combined behaviors of the node components. We will consider mainly the first task, as the second is more meaningful when the focus is on reasoning about achieving goals.

The requirements for a PAGODA node coordinator (§4.3) are that a message is delivered to a component only if the component is enabled for that message, and that certain cross component message ordering requirements are met. A natural way to formalize the component enabledness criteria given in §4.3 is in terms of messages sent and delivered: a message of one kind has been delivered (request) and no message of a corresponding kind (reply) has been sent, or that a message of one kind has been sent (request) and no message of another (reply) has been delivered. The cross-component message ordering constraints can be formalized as equality of subsequences of delivered messages. In particular, the formalization talks about events (message send / delivery), and not about system state. Using temporal logics such as LTL or CTL one can specify properties of system states but on can not directly talk about transitions or events. There are workarounds such as adding history components to the state and modifying the rules to record events they generate, for example each component could have a history message queue that records the sequence of inputs and outputs at that component. Mechanical verification of such properties requires something beyond model-checking, either developing special strategies to guide a general purpose prover, or inventing abstractions that reduce the problem to something amenable to model checking. Other logics such as process logics ([6,7]), or action logics such as Lamport's TLA [19] could be considered, as the actions or labels could be used to represent events. The process logics typically don't treat labels / data with structure and again the formalization would not be very natural. TLA (or TLA+) is quite expressive and might be suitable. Being a general purpose logic, some work would be needed to

formalize the full coordinator RRD model.

We propose an alternative approach, namely to develop special purpose logics that are tailored the structure common to wide classes of policy-based coordinator systems. The objective is to raise the level of abstraction of reasoning. Towards this end, one can begin to identify common classes of requirements which can be verified by easily checkable criteria on coordinator policies. The properties and rules for checking should evolve into a (policy-based) logic with a proof system based on such rules, whose models are policy based coordinator systems. The PAGODA node requirements present two such classes. These could be thought of as a first step towards a policy based logic.

As an example, to show that components L and KB see update messages and logged messages in the same order (and the same messages) it is sufficient to check that

1. if `M` = `msg(KB,x,Update)` then
   `enabled(P,w,M)` implies `msg(L,-,notify(M))` is in `deliverM(P,w,M)` iff
   `M` is in `deliverM(P,w,M)`
2. if `M` = `msg(KB,x,MB)` and `MB` has type `Goal`, `Set*`, or `SensorReply` then
   `enabled(P,w,M)` implies `msg(L,-,notify(M))` is in `deliverM(P,w,M)` iff
   `msg(KB,-,log(M))` is in `deliverM(P,w,M)`

Notice that 1. reduces to checking that `M` = `msg(KB,x,Update)` implies `msg(L,-,notify(M))` is in `deliverM(P,w,M)`; and for 2. the enabledness assumption is only needed for `SensorReply` messages.

For synchronization constraints it is sufficient to show that the wait4 set tracks enableness conditions for messages and is correctly used to enable delivery. For example, if X is only enabled for `SenseReply` messages in response to `Sense` messages it sent, it suffices to check that

`msg(X,y,?sense)` is in `deliverM(P,w,M)` implies
`w4(msg(y,X,?SenseReply), iMs, xMs)` in `updWait4s(P,w,M)`

for some `iMs`, `xMs`, and that

`msg(X,y,?senseReply)` is in `deliverM(P,w,M)` implies
`w` = `w'` `w4(msg(X,y,?SenseReply), iMs, xMs)`

for some `iMs`, `xMs`, `w'`, and furthermore an element of the wait4 set of the form

`w4(msg(X,y,?SenseReply), iMs, xMs)`

is removed if and only if a message of the form `msg(X,y,?senseReply)` is in `deliverM(P,w,M)`.

Similarly for other reply enabledness constraints.

For serialized requests such as the reasoner 'one goal at a time' constraint it suffices to check

`msg(R,z,Goal)` in `deliverM(P,w,M)` implies
`w4(msg(z,R,ack(Goal)), iMs, xMs)` in `updWait4s(P,w,M)`

for some `iMs`, `xMs`, and

`msg(R,y,Goal)` in `deliverM(P,w,M)` implies
`w` contains no element of the form `w4(msg(z,R,ack(?Goal)),iMs,xMs)`.

Furthermore, an element of the wait4 set of the form `w4(msg(X,y,ack(?Goal)), iMs, xMs)` is removed if and only if a message of the form `msg(X,y,ack(?goal))` is in `deliverM(P,w,M)`.

5 Distributed Coordination

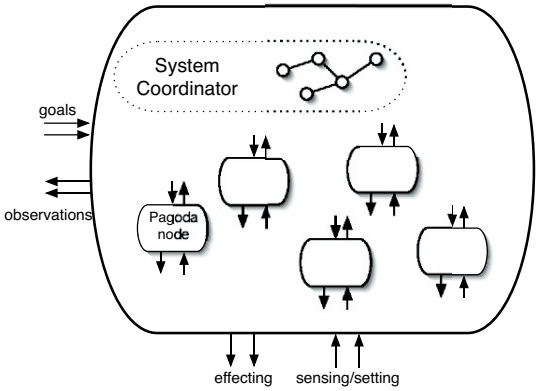

Fig. 3. PAGODA system architecture

Figure 3 shows the structure of a distributed PAGODA system. It contains multiple PAGODA nodes whose interactions are controlled by a system coordinator. From the outside, this differs from a PAGODA node in that there are multiple asynchronous points of interaction —for goals coming in to different nodes and points of observation of the effects. From the inside, the system coordinator is responsible for coordination amongst the PAGODA nodes rather than amongst components of a given node, and thus is concerned with different issues such as dissemination of local knowledge to form a more global picture, negotiation of tasks, or synchronization of actions to carry out some task. The system coordinator is also policy based. It has an abstract model of the nodes it is coordinating, for example the potential connectivity between nodes. An important difference in designing a system coordinator is the challenge of designing policies that can be distributed, thus turning the system coordinator into distributed collection of coordinators, one on each node, that require only minimal communication amongst themselves to achieve the effects of the system coordinator.

In an ongoing project concerning ad hoc wireless radio networks, methods for specifying distributed coordinators are being developed based on a logical approach to distributed monitoring [33] and a distributed AI approach to distributed problem solving [20,21]. In this application, a key coordinator responsibility is controlled dissemination of information. Here policy determines what information to propagate, when and to which nodes. Another coordination responsibility is distributed management of resources (bandwidth, energy, data storage). Here we have an interesting situation in which any realization of the coordinator must rely on the

resources it manages to carry out the management process! To manage scale, we propose a dynamically changing hierarchy of organizational units formed according to constraints defined by goals and policies, and possibilities according to available capabilities. This is a significant challenge for the design of distributable coordination policies.

6 Related Work

As mentioned in § 1, the starting point for PAGODA was a study of the Mission Data System architecture for autonomous space systems [15]. A key feature of MDS is goal and model based operation. System state is structured as a collection of *state variables* shared by all system components. A goal is a constraint on a state variable over some time interval. Each state variable has an associated goal achiever compose of a goal elaborator that decides what actions to execute, a controller to issue commands, a sensor to read raw sensor data, and an estimator to interpret the sensor data. A scheduler serves as coordinator, executing components according to a fixed schedule. This provides predictability at the cost of flexibility. MDS is intended as an architecture for a single node with real-time concerns and largely synchronous interactions. There is no mechanism for treating asynchronous distributed systems. Goal elaboration is a planning activity rather than based on logical reasoning and models tend to be buried in code rather than declaratively specified and kept in a shared knowledge base. The state variable approach has some advantages but relations between state variables are complicated to manage.

There are numerous languages for specifying or programming coordination whose semantics has been studied by a variety of techniques. The approach of the work presented here is to start with a semantic model of distributed object coordination, focusing on interactions rather than system state, and study language independent coordination mechanisms, their semantic consequences, and semantics-based reasoning principles.

Tuple space languages include Linda [18] and its mobile extension, Lime [28]. The Klaim[27] language combines process-algebra and tuple spaces. StoKLAIM [12] is a Markovian extension of KLAIM that supports distribution awareness and dynamic system architecture configuration.

Actor languages with coordination abstractions include Synchronizers [17,16] and Real-Time Synchronizers [30]. These languages provide linguistic constructs for specifying coordination and come with compilation transformations for implementation in terms of standard actors.

The Actor-Role-Coordinator (ARC) model [31] has several similarities to our policy-based coordinators—our objects are very similar to actors, and the coordination constraints are separated from the actor behavior and simply manage message delivery. The ARC model is concerned with time-related quality-of-service constraints, while PAGODA currently treats time at the level of an abstract partial order of events.

Reo [2,3] is a channel based coordination model where complex coordinators

called connectors are constructed by composing smaller one. Like the RRD coordination model, REO coordination is achieved by controlling the communication semantics. A semantic model based on timed data streams and co-inductive reasoning principles is given in [1].

The Mobile Unity language provides coordination primitives as well as a logic for reasoning about Mobile Unity specifications. Refinement from a high-level logical specification to mobile unity code is illustrated in [32]. Coordination properties are based on system state rather than interaction events.

7 Summary and Future Work

We have presented a case study using policy-based coordination as the basis of a flexible modular architecture for interactive autonomous agents called PAGODA. The formal specification of PAGODA node coordinators in the Reflective Russian Dolls (RRD) framework was described, and principles for reasoning about two simple classes of coordinator requirements were presented. Issues in designing distributed system coordination policies were discussed and contrasted to local node level coordination. As indicated in §5 work in is progress to address these issues.

Additional case studies are needed to continue to develop useful abstract classes of coordinator requirements and associated policy properties that can be checked (in the equational theory defining the policy) that ensure requirements are met.

An interesting future task is to make a detailed comparison with Reo, in particular a rewriting semantics for REO possibly using the RRD framework; a study of classes of policies that correspond naturally to Reo connectors; and developing more general mappings of RRD coordinators to Reo. The compositional nature of Reo could be beneficial in developing policy-based logics for coordination. In addition recent work on probabilistic semantics for Reo could be a path to extending the RRD model to treat QoS requirements [4]. Another interesting source of ideas for probablistic / stochastic extension is recent work on probablistic and stochastic extensions of KLAIM [29,12,11].

Acknowledgement

Thanks to the PAGODA team who developed the executable prototype. The team discussions were instrumental in clarifying the roles and interactions of different components. The author also wishes to thank Daniel Elenius and the anonymous reviewers for helpful criticisms.

References

[1] F. Arbab and J. J. M. M. Rutten. A coinductive calculus of component connectors. Technical Report SEN=R0216, Centrum voor Wiskunde en Informatica (CWI), 2002.

[2] Farhad Arbab. A chanel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004. Also Centrum voor Wiskunde en Informatica report SEN=R0203.

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

[4] C. Baier. Stochastic semantics for reo connector circuits, 2005. submitted for publication.

[5] S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based constraint satisfaction and optimization. *Journal of the ACM*, 44(2):201–236, 1997.

[6] Luís Caires and Luca Cardelli. A spatial logic for concurrency (part i). *Information and Computing*, 186(2):194–235, 2003.

[7] Luís Caires and Luca Cardelli. A spatial logic for concurrency - ii. *Theoretical Computer Science*, 322(3):517–565, 2004.

[8] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and C. Talcott. Maude 2.0 Manual, 2003. http://maude.cs.uiuc.edu.

[9] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. The Maude 2.0 system. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications (RTA 2003)*, volume 2706 of *Lecture Notes in Computer Science*, pages 76–87. Springer-Verlag, 2003.

[10] Manuel Clavel and José Meseguer. Reflection in Conditional Rewriting Logic. *Theoretical Computer Science*, 285:245–288, 2002.

[11] R. De Nicola, J.-P. Katoen, D. Latella, and M. Massink. StoKlaim: A stochastic extension of Klaim. Technical Report ISTI-2006-TR-01, ISTI CNR, Italy, 2006.

[12] R. De Nicola, J.P. Katoen, D. Latella, and M. Massink. Towards a logic for performance and mobility. In *3rd Workshop on Quantitative Aspects of Programming Languages, QAPL05*, pages 132–146. University of Edinburgh (LFCS), 2005.

[13] G. Denker and C. L. Talcott. Formal checklists for remote agent dependability. In *Fifth International Workshop on Rewriting Logic and Its Applications (WRLA'2004)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2004.

[14] G. Denker and C. L. Talcott. A formal framework for goal net analysis. In *Workshop on Verification and Validation of Planning Systems*. AAAI, 2005.

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

[16] S. Frølund. *Coordinated Distributed Objects: An Actor Based Approach to Synchronization*. MIT Press, 1996.

[17] S. Frølund and G. Agha. A language framework for multi-object coordination. In *ECOOP 1993*, volume 707 of *Lecture Notes in Computer Science*. Springer, 1993.

[18] David Gelernter. Generative communication in linda. *TOPLAS*, 7(1):80–112, 1985.

[19] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Weseley, 2002.

[20] Roger Mailler and Victor Lesser. Solving Distributed Constraint Optimization Problems Using Cooperative Mediation. In *Proceedings of Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004)*, pages 438–445. IEEE Computer Society, 2004.

[21] Roger Mailler and Victor Lesser. Using Cooperative Mediation to Solve Distributed Constraint Satisfaction Problems. In *Proceedings of Third International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS 2004)*, volume 1, pages 446–453, New York, 2004. IEEE Computer Society.

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

[23] J. Meseguer and C. L. Talcott. Semantic models for distributed object reflection. In *European Conference on Object-Oriented Programming, ECOOP'2002*, volume 2374 of *Lecture Notes in Computer Science*, pages 1–36, 2002. invited paper.

[24] José Meseguer. A rewriting logic sampler. In Dang Van Hung and Martin Wirsing, editors, *Second International Colloquium on Theoretical Aspects of Computing (ICTAC 2005)*, volume 3722 of *Lecture Notes in Computer Science*, pages 1–28. Springer, 2005.

[25] N. Muscetolla, P. Pandurang, B. Pell, and B. Williams. Remote Agent: To Boldly Go Where No AI System Has Gone Before. *Artificial Intelligence*, 103((1–2)):5–48, 1998.

[26] R. De Nicola, G. Ferrari, U. Montanari, R. Pugliese, and E. Tuosto. A process calculus for qos-aware applications. In *Seventh International Conference on Coordination Models and Languages*, 2005.

[27] R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: A kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.

[28] G. Picco, A. Murphy, and G.-C. Roman. LIME: Linda meets mobility. In *21 Int. Conf. on Software Engineering*, pages 368–377, 1999.

[29] A. Di Pierro, C. Hankin, and H. Wiklicky. Probabilistic klaim. In *Proceedings COORDINATION 2004*, 2004.

[30] Shangping. Ren. *An Actor-Based Framework for Real-Time Coordination*. PhD thesis, University of Illinois at Urbana-Champaign, 1997.

[31] Shangping Ren, Yue Yu, Nianen Chen, Kevin Marth, Pierre-Etienne Poirot, and Limin Shen. Actors, roles and coordinators: A coordination model for open distributed and embedded systems. In *Coordination 2006*, 2006. to appear.

[32] Gruia-Catalin Roman, Christine Julien, and Qingfeng Huang. Formal specification and design of mobile systems. In *Formal Methods for Parallel Programming: Theory and Applications*, 2002.

[33] K. Sen, A. Vardhan, G. Agha, and G. Rosu. Efficient decentralized monitoring of safety in distributed systems. In *Proceedings of 26th International Conference on Software Engineering (ICSE'04)*, pages 418–427. IEEE, May 2004.

[34] Carolyn 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.

[35] Martin Wirsing, Grit Denker, Carolyn Talcott, Andy Poggio, and Linda Briesemeister. A rewriting logic framework for soft constraints, 2006. submitted to WRLA06.