A CSP-Based Agent Modeling Framework for the Cougaar Agent-Based Architecture

Denis Gračanin, H. Lally Singh
Department of Computer Science
Virginia Tech
Blacksburg, VA 24061, USA
{gracanin, lally}@vt.edu

Michael G. Hinchey
NASA Goddard Space Flight Center
Information Systems Division
Greenbelt, MD 20771, USA
Michael.G.Hinchey@nasa.gov

Mohamed Eltoweissy
Department of Computer Science/NVD
Virginia Tech
Falls Church, VA 22043, USA
toweissy@vt.edu

Shawn A. Bohner
Department of Computer Science
Virginia Tech
Blacksburg, VA 24061, USA
sbohner@vt.edu

Abstract

Cognitive Agent Architecture (Cougaar) is a Java-based architecture for large-scale distributed agent-based applications. A Cougaar agent is an autonomous software entity with behaviors that represent a real-world entity (e.g., a business process). A Cougaar-based Model Driven Architecture approach, currently under development, uses a description of system's functionality (requirements) to automatically implement the system in Cougaar. The Communicating Sequential Processes (CSP) formalism is used for the formal validation of the generated system. Two main agent components, a blackboard and a plugin, are modeled as CSP processes. A set of channels represents communications between the blackboard and individual plugins. The blackboard is represented as a CSP process that communicates with every agent in the collection. The developed CSP-based Cougaar modeling framework provides a starting point for a more complete formal verification of the automatically generated Cougaar code. Currently it is used to verify the behavior of an individual agent in terms of CSP properties and to analyze the corresponding Cougaar society.

1. Introduction

Agent-based systems provide a foundation for the development of large-scale applications such as logistics management, battlefield management, supply-chain management, to mention just a few. An example of agent-based systems is Cognitive Agent Architecture (Cougaar) [1, 2]. Cougaar provides a software architecture for distributed agent-based applications in domains characterized by hierarchical decomposition, tracking of complex tasks, generation and maintenance of dynamic plans.

The ability to develop very complex applications comes at a price. It takes a lot of effort and learning in order to have complete understanding and the ability to effectively use such agent-based systems. Cougaar's capabilities and the complexity of the systems implemented in Cougaar bring additional demands for analysis, testing, and verification. Existing techniques can be adapted to deal with these demands.

The Model Driven Architecture (MDA) approach [15] can be used for developing applications using the Cougaar agent-based architecture. The Cougaar MDA (CMDA) provides automated software-artifact generation and simplifies Cougaar-based application development by providing two abstraction layers. Requirements are translated into the General Domain Application Model (GDAM) components. The GDAM components are then translated into the General Cougaar Application Model (GCAM) components which are used to automatically generate the Cougaar code (software applications) [8].

As a first step for the formal verification of the generated Cougaar-based system, a formal model is used to verify the behavior of a Cougaar agent. Single-agent and multi-agent communications models provide a framework for the formal verification of requirements specifying a Cougaar-based application and its properties. Hoare's Communicating Sequential Processes (CSP) formalism [5, 12, 18] is used to develop a formal model of a Cougaar agent and a simple CSP model of a Cougaar-based system.

The remainder of the paper is organized as follows. Section 2 briefly describes Cougaar and its capabilities. Section 3 describes approaches to the modeling of agent-based systems. Section 4 discusses the CSP-based Cougaar model. Section 5 provides an example, while Section 6 concludes the paper.

2. Cougaar Agent-Based Architecture

Cougaar is a large-scale workflow engine built on a component-based distributed agent architecture [1]. It is deployed as a society of agents, which communicate and work together to solve a problem. A Cougaar society is a set of agents running on one or more interconnected computers, all working together to solve a common class of problems. The problem may be partitioned into sub-problems, in which case the responsible subset of agents is called a community. A society may have one or more communities. The relationship between societies, communities, and agents is not a strict one; a society may directly contain both agents and communities. While a society has a real-world representation, a set of computers running a Cougaar system, a community is only notational in nature.

Some recent efforts in addressing Cougaar architectural characteristics include performance and survivability enhancements. There are many different requirements on the metrics collection and communication channels [10]. Performance improvements and measurement require the use of multiple communication channels to improve overall performance with some data duplication overhead. A general approach for dynamically improving overall system survivability for the large class of applications is described in [9]. Constraints include finite resources, multiple dimensions of success, and global optimization of the goal.

2.1. Agent

A Cougaar agent is a first-class member of a Cougaar Society [1] and it contains a Blackboard and one or more Plugins. While the specific purpose of any agent is chosen by the system developer, the objective is for a single agent to represent a single organizational entity or part of that entity.

At the most basic level, an agent consists of two parts: a Blackboard and a set of Plugins (Figure 1). The former is a container of objects, with a subscription-based change notification mechanism; the latter is a set of responders to these notifications, with the ability to change the contents of the Blackboard.

The Blackboard serves as the communications backbone connecting the Plugins together. More importantly, it serves

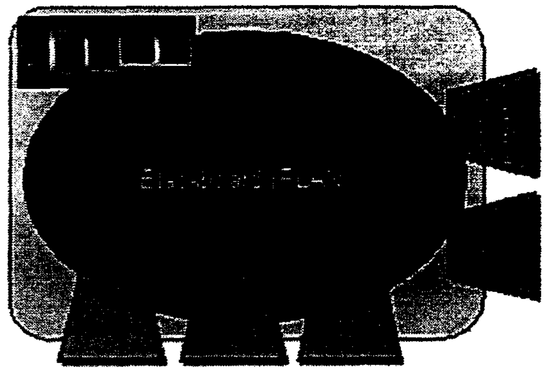

Figure 1. Cougaar Agent Structure [1]

as the entry point for any incoming messages to the agent as a whole, which are then picked up by the Plugins for handling. All instance-specific behavior of the agent is implemented within the Plugin. A Plugin listens to *add*, *remove*, and *change* events on the Blackboard. Evaluating the objects involved in the event, the Plugin may respond by performing some computation, changes to the Blackboard, or some external work.

A Cougaar Node conceptually encapsulates a set of agents. Agents can collaborate with other agents in the same Node or with agents in other Nodes. However, it is not a direct collaboration. Instead, Cougaar Tasks are allocated to Cougaar Organizations, which are representations of agents in the local Blackboard. The subscription mechanism allows agents to use Tasks to exchange messages (objects). The Cougaar communication infrastructure then ensures that the Task is sent to the destination Organization's (i.e. agent's) Blackboard.

2.2. Multi-Agent Communications

Cougaar's multi-agent communications system is quite sophisticated. There are several ways for agents to communicate with each other. The methods are mostly based upon a Plugin, be it a specialized or a library component, observing one agent's Blackboard and propagating relevant data to another.

2.3. Cougaar Model Driven Architecture

The MDA approach advocates converting a Platform Independent Model (PIM) into a Platform Specific Model (PSM) through a series of transformations, where the PIM is iteratively made more platform-specific, ending in the PSM. The PIM is used to represent system's functionality without including any technical aspects (Table 1).

The encoding of requirements is done in stages using the "separation of concerns" principle. First, the workflow of the intended system needs to be drawn up. Then, the required domain components are retrieved from the repository, the parameters for the components are provided and the components are linked to obtain the domain model of the system. The platform-independent model is represented in the General Domain Application Model (GDAM) layer, while the General Cougaar Application Model (GCAM) layer provides the platform-specific model. Once the domain model is validated, the developers can check and ensure that the GCAM components required to implement the domain model are present and linked properly. The linked GCAM components represent the application model of the intended system. Developers also ensure that the requirements captured in the domain level have been transformed into application parameters properly. Once the domain and application models are properly built, the user can instruct the system to convert the models into required software artifacts including Requirements, Design, Code and Test cases. The transformation of the models into software artifacts is handled by the compiler of the Cougaar Model Driven Architecture (CMDA) system. Once the compilation is complete, the user is informed of errors and warnings. Hence the CMDA system deals only with requirements collection from the user and allows users to verify the requirements that have been properly transformed into application model parameters.

Computationally Independent Model (input requirements)
Platform Independent Model (GDAM)
Platform Specific Model (GCAM)

Table 1. Basic CMDA Approach

The following goals and priorities were formulated in order to develop a tool based on the proposed approach [8]:

- Fully automated software artifacts (requirements, design document, code, and test cases) generation is a desirable goal.
- The generated requirements are partial in nature.
- The development of tools and implementation mechanisms are of lower priority than formulating the "recipes" for transformations.

3. Modeling Agent-Based Systems

Large agent-based systems usually operate in a distributed manner and depend on the underlying communication infrastructure designed for a specific system. Sound theoretical concepts can provide solutions for distributed multi-agent systems that are superior to "usual" ad-hoc implementations [3].

Suitability of agent modeling techniques to agent-based systems development is discussed in [20]. It provides directions for research and development of agent-oriented modeling techniques. A set of criteria is proposed, based on the desired characteristics, for appropriate modeling techniques and approaches.

The Specification Language for Agent-Based Systems (SLABS) [19] is based on a meta-model of multi-agent systems. An agent-based system is modeled at two levels. The macro-level describes the relationships among agents. The micro-level describes the behaviors of individual agents. A diagrammatical notation includes collaboration, behavior, and scenario diagrams.

Petri nets [7, 16] are used extensively in modeling, simulation and analysis of distributed and multi-agent systems. Predicate/transition nets, a form of high-level Petri nets, can be used to construct a multi-agent model where transitions represent agent capabilities [24]. Analysis of Petri net properties like reachability, liveness, etc., are used to verify that the multi-agent plans indeed achieve their goals.

DESIRE is a framework for compositional systems (including multi-agent systems) [4]. It uses hierarchical task decomposition and supports specification of agents that integrates reasoning with interactions (communication, observation and actions execution) and executive autonomy. The semantic approach based on temporal logic is outlined and applied for the case of compositional multi-agent systems.

The TYPELAB system provides a reflective architecture for formalizing and reasoning about artifacts created during a software development process [17]. Object- and meta-level reasonings are combined using reflection principles to generate "standard" units.

The CSP formalism uses a simple event-based description and algebra for the description and analysis of systems [11, 12]. Originally designed for Transputers [23], it provides a solid theoretical foundation for modeling software system behavior and formally proving properties.

The core CSP components are processes and events. An event is an atomic, named entity. A process is a definition of allowable sequences of events. It is defined as a set of events with arrows ($\rightarrow$) between them.

Processes may use recursion, conditionals, states, and references to other processes to define these sequences. A process' state may be shown as a subscript. Processes may be composed together using the parallel operator ($\|$), such that the actual event that occurs determines which of the processes executes.

Processes communicate through fixed, named, unidirectional *channels*. Such channels have exactly one fixed input process, and exactly one fixed output process. The act of sending a message over the channel is actually an event with two different names in the two processes. An output event is shown as a channel name followed by an exclamation point, followed by the message being sent. For example *out!m* sends the message *m* over the channel *out*. Input events use the similar notation, with a question mark replacing the exclamation point.

CSP can be extended, for example, to include priority specifications [6]. Appropriate algorithms can be developed to analyze properties of distributed systems that use a CSP-like communication infrastructure [13]. CSP models for a specific programming language implementations further increase modeling capabilities (e.g. for Java [22]).

4. CSP Framework for Cougaar-based systems

The control architecture of the Cougaar distributed agent system [14] consists, at the node level, of the following components:

- Operating mode: a data structure for control inputs.
- Conditions: generalized sensor input information.
- Plays: control laws (restrictions on operating modes and conditions).
- Playbook: a list of plays tested in sequence for current conditions.
- Playbook manager: a component maintaining and manipulating a playbook.
- TechSpecs: a high-level description of components behavior.
- Adaptivity engine: a controller for the agent and some other external components.
- Operating mode policy: higher-level system policy.

This control architecture can be easily adopted for various applications. Given the characteristics of an agent-based system, it is difficult to apply conventionally control theory methods directly [14]. Large-scale parallelization and scalability is achieved by using a multi-tier communication infrastructure where each tier uses different communication techniques [21]. The CSP-based Cougaar modeling framework can provide a formal framework to analyze behavior of a single agent and its interactions with other agents.

4.1. Single Agent

There are two assumptions used in developing the CSP model of an agent:

- The Plugins attached to a given agent are fixed. All Plugins are known at the start and the set cannot change over time. However, a Plugin may decide to start or stop participating with the agent it is connected to at any time.
- A subscription mechanism is replaced by a broadcast mechanism. Each Plugin is notified of every *add*, *change*, and *remove* occurring on its Blackboard.

These assumptions may affect the accuracy and efficiency of the systems. The model accuracy can be preserved by introducing a *UnaryPredicate* that filters Blackboard events. Upon reception of a Blackboard event, the Plugin can evaluate its *UnaryPredicate*, and ignore the event if the predicate returns *false*. If the actual Plugin is only subscribed at certain times, we model its subscription state as a state variable s. Then, the *UnaryPredicate* can simply run with an additional requirement, that $s = true$. Reduced efficiency, while important for deployment, is not relevant for modeling the system.

An agent is defined as:

$$A = < B_O, P >$$

where B_O is the Blackboard, O is the set of objects, $P = \{p_1, p_2, \dots p_N\}$ is the set of Plugins, and N is the number of plugins in P.

4.2. Communication

Before getting into the details of the agent's components, it is necessary to describe the communications channels between the Blackboard and the Plugins. The Blackboard defines N input channels $(\beta_1, \dots, \beta_N)$. These channels accept input from the respective Plugins $(p_1, \dots, p_N)$. The Blackboard sends its notifications to the Plugins via the respective channels $(\rho_1, \dots, \rho_N)$.

4.3. Blackboard

The Blackboard contains objects, which are modeled as an embedded pairs.

$$object = < types, attributes >$$
$$types = \{t_1, t_2, \dots\}$$
$$attributes = \{< name_1, value_1 >, < name_2, value_2 >, \dots\}$$

The *object* represents any arbitrary object that can be put on the Blackboard. In the actual system, this is an instance of a Java class. The class is always a descendant of the *Object* class. The *object* is an ordered pair of *types* and a *attributes*. The *types* is the set of names, including the object's real type, and the names of all supertypes. The *types* always contain the element *Object* due to Java's inheritance model where the *Object* class is the root of the Java class hierarchy. The *attributes* is the set of object attributes represented as $< name, value >$ pairs. The set of all possible objects is defined as $\mathcal{O}$.

The Blackboard is modeled as the CSP Process B_O with the state $O = \{o_1, o_2, \ldots\}$ representing the set of the current Blackboard objects. O is always a subset of $\mathcal{O}$. The alphabet of the Blackboard's CSP process B_O includes symbols *add*, *change*, *remove*, and all the objects in $\mathcal{O}$.

$$\alpha B_O = \left\{ \begin{array}{l} add, \\ change, \\ remove, \\ o : o \in \mathcal{O} \end{array} \right\}$$

The subprocesses of the Blackboard, *Add*, *Chg*, *Rem*, and *Notify* are shown in Figure 2. *Add*, *Chg*, and *Rem* take in the object o from the same input channel, update O as needed, and call *Notify* to notify the Plugins. *Notify*'s definition is simple, it sends the event e (be it *add*, *change*, or *remove*) and the object o to all the Plugins via their input channels.

```
process
  $B_O =$
    let
      process
        $Add(O, i) =$
          $\beta_i?o \rightarrow$
            if $o \notin O$ then $O := O \cup \{o\}$
            $\rightarrow Notify(add, o)$
            else $STOP$
        $Chg(i) =$
          $\beta_i?o \rightarrow$
            if $o \notin O$ then $STOP$
            else $Notify(change, o)$
        $Rem(O, i) =$
          $\beta_i?o \rightarrow$
            if $o \in O$ then $O := O \setminus \{o\}$
            $\rightarrow Notify(remove, o)$
            else $STOP$
        $Notify(e, o) =$
            $\forall i \in N, (\rho_i!e \rightarrow \rho_i!o) \rightarrow B_O$
    $\|_{i=1}^{N} \beta_i?cmd \rightarrow$
      if $cmd = add$ then $Add(O, i)$
      if $cmd = change$ then $Chg(i)$
      if $cmd = remove$ then $Rem(O, i)$
      else $STOP$
```

Figure 2. Blackboard

4.4. Plugin

A Plugin p_i is a CSP process that:

1. Belongs to an agent.
2. Listens on an input channel ρ_i for events.
3. Decides if each event is relevant.
4. If relevant, acts in response to it, possibly resulting in additional events on the Blackboard.

The first and second steps have already been described. The third step is essentially a model of the Cougaar *UnaryPredicate*. As shown in the Blackboard model, a CSP process can use conditional statements. An example of a simple unary predicate is shown in Figure 3. and the corresponding CSP model is shown in Figure 4.

The subprocess $Pred(o)$ acts as a predicate, proceeding forward when the object matches, recursing back to p_i when not. The other subprocess, $Act(o)$ is the response action to a satisfied predicate. In this case it simply posts a *String* with value `"PAID"` to the Blackboard.

4.5. Multiple Agents

The support for multiple agents and their interactions is provided by the *middleman* that maintains channels with all the Plugins in the system, and relays messages to a new input channel for each Blackboard. Each Agent, Plugin, and Blackboard are uniquely named within the global scope. That is achieved by adding another subscript, a, to every identifier. This subscript identifies the Agent/Blackboard. The *middleman*'s channels coming in from each plugin in the system would be $\mu_{a,i}$ corresponding to the channel's source, the Plugin $p_{a,i}$. The channels going out from the *middleman* to each Blackboard are named $\beta_{a,0}$.

5. Example

The Dining Philosophers problem [16] is used as an example. In this problem, there is a circular table with D seats, D forks, and with a bowl of spaghetti in the middle. At random intervals, any one of D philosophers sits down, picks up the forks on the left and right, consumes spaghetti for a random amount of time, puts down the forks, and stands up again. If all D philosophers sit down at once, pick up the fork on their left, and then try to pick up the right fork, they will deadlock waiting for the philosopher on their

```
UnaryPredicate clearPaymentPredicate =
  new UnaryPredicate() {
    public boolean execute(Object o) {
      if (o instanceof Task) {
        Task task = (Task) o;
        if ((task.getVerb() != null) &&
         task.getVerb().toString().equals(
          BolSocietyUtils.PAYMENT_VERB)) {
          return true;
          }
      }
      return false;
    }
  };
```

Figure 3. *UnaryPredicate* code

```
process
  $p_i =$
    let
      process
        $Pred(o) =$
          if $Task \in o.types$ and
          $< verb, PAYMENT\_VERB >$
            $\in o.attributes$
          then $SKIP$
          else $p_i$
        $Act(o) =$
          $\beta_i!add \rightarrow$
          $\beta_i! < \{Object, String\},$
                $\{< value, PAID >\} > \rightarrow SKIP$
    $\rho_i?e \rightarrow \rho_i?o \rightarrow$
      if $e = add$ then $Pred(o) \fatsemi Act(o) \fatsemi p_i$
      else $p_i$
```

Figure 4. Example Plugin

right to put down the forks. To illustrate the model's use, a CSP/Cougaar model of the problem is provided, followed by the description how the deadlock occurs.

Three CSP abstraction processes are used as subroutines to add, remove, and listen for objects on the Blackboard. The listener removes the object it founds (Figure 5).

5.1. Forks

Figure 6 shows that reservation and allocation subprocesses are executed repeatedly. The reservation subprocess waits for a *Request* object to be placed on the Blackboard, specifying a *get* of the fork i. The allocation process posts a *Response* object to the Blackboard, mirroring the parameters of the *Request*.

```
process
  $Post(o) = \beta_n!add \rightarrow \beta_n!o \rightarrow SKIP$
```

```
process
  $Remove(o) = \beta_n!remove \rightarrow \beta_n!o \rightarrow SKIP$
```

```
process
  $Listen(o) = \rho_n?e \rightarrow \rho_n?p \rightarrow$
    if
      $o.types \in p.types$
      and$o.attributes \in p.attributes$
    then $Remove(o)$
    else $Listen(o)$
```

Figure 5. Abstraction Processes

5.2. Philosophers

All a philosopher does is to repeatedly get the two forks and release them in left-to-right order. The philosopher's sitting, standing, and eating states are not relevant for the deadlock problem. As shown in Figure 7, there are two subprocesses for getting and releasing the forks (% denotes the modulus operator).

5.3. Deadlock

A deadlock state of the system is described by providing an ordered list of events that prevents any subsequent event. First two philosophers and two forks are shown in Figure 8. Only the relevant events are described; add events of requests or responses that try to get or to release a fork. Each philosopher successfully gets a left fork. The first attempt to acquire a right fork will cause deadlock. It is important to note that releases of forks are not ignored; none actually occurs here. To prove that a system does not deadlock is far more difficult. CSP provides rewriting rules and equivalence relations to help analyze a model for a specific property.

```
process
  $FORK_i =$
    let
      process
        $Reserve(i) =$
          $Listen($
            $< \{Request\},$
            $\{< id, i >, < act, get >\} >$
          $)$
          $\fatsemi\ Post($
            $< \{Object, Response\},$
            $\{< id, i >, < act, get >\} >$
          $)$
        $Allocate(i) =$
          $Listen($
            $< \{Request\},$
            $\{< id, i >, < act, release >\} >$
          $)$
          $\fatsemi\ Post($
            $< \{Object, Response\},$
            $\{< id, i >, < act, release >\} >$
          $)$
    $Reserve(i) \fatsemi Allocate(i) \fatsemi FORK_i$
```

Figure 6. Forks

```
process
  $p_i =$
    let
      process
        $Get(i, s) =$
          $Post($
            $< \{Object, Request\},$
            $\{< id, i >, < act, get >,$
            $< side, s >\} >$
          $)$
          $\fatsemi\ Listen($
            $< \{Response\},$
            $\{< id, i >, < act, get >\} >$
          $)$
        $Release(i, side) =$
          $Post($
            $< \{Object, Request\},$
            $\{< id, i >, < act, release >,$
            $< side, s >\} >$
          $)$
          $\fatsemi\ Listen($
            $< \{Response\},$
            $\{< id, i >, < act, release >\} >$
          $)$
    $Get(i, left) \fatsemi Get((i + 1)\%N, right)$
    $\fatsemi\ Release(i, left)$
    $\fatsemi\ Release((i + 1)\%N, right) \fatsemi p_i$
```

Figure 7. Dining Philosophers

```
    $failure =$
(   $add, < \{Request\},$
     $\{< id, 1 >, < act, get >, < side, left >\} >,$
    $add, < \{Response\}, \{< id, 1 >, < act, get >\} >$
    $add, < \{Request\},$
     $\{< id, 2 >, < act, get >, < side, left >\} >,$
    $add, < \{Response\}, \{< id, 2 >, < act, get >\} >,$
    $add, < \{Request\},$
     $\{< id, 2 >, < act, get >, < side, right >\} >$
    $\ldots)$
```

Figure 8. Deadlock state

6. Conclusions

Hoare's CSP can be used as a formalism for modeling and analyzing distributed multi-agent systems. A basic CSP model of a Cougaar system can be generated in parallel with the CMDA generated Cougaar code. It provides a model for analyzing the interactions of agents, their Plugins, and Blackboards. A CSP tool is then used to detect deadlocks or to determine other CSP properties.

While the described approach simplifies some characteristics of the Cougaar agents and related components, it is a stepping stone towards a more complete analysis. Current research focuses on completing the CMDA development where the major focus is on the formal validation of the generated Cougaar code.

Acknowledgements

This work has been supported, in part, by the DARPA STTR grant "AMIIE Phase II — Cougaar Model Driven Architecture Project," (Cougaar Software, Inc.) subcontract number CSI-2003-01. We would like to acknowledge the efforts, ideas, and support that we received from our research team including Todd Carrico and Tim Tschampel.

Part of this work was supported by the NASA Goddard Space Flight Center Technology Transfer Office. Denis Gračanin was supported by an ASEE/NASA Summer Faculty Fellowship hosted at the NASA Software Engineering

Laboratory (Code 581), NASA Goddard Space Flight Center.

References

[1] Cougaar architecture document. Technical report, BBN Technologies, 5 July 2004. Version for Cougaar 11.2.

[2] Cougaar developers guide. Technical report, BBN Technologies, 5 July 2004. Version for Cougaar 11.2.

[3] J. W. Amtrup and J. Benra. Communication in large distributed AI systems for natural language processing. In *Proceedings of the 16th Conference on Computational Linguistics*, pages 35–40. Association for Computational Linguistics, 1996.

[4] F. Brazier, P. van Eck, and J. Treur. Design of a modelling framework for multi-agent systems. In R. Albrecht and H. Herre, editors, *Trends in Theoretical Informatics*, volume 89 of *Schriftenreihe der Österreichischen Computer Gesellschaft*, pages 173–191. R. Oldenbourg, Wien, 1996.

[5] J. Davies. *Specification and Proof in Real-Time CSP*. Distinguished Dissertations in Computer Science. Cambridge University Press, Cambridge, 1993.

[6] C. J. Fidge. A formal definition of priority in CSP. *ACM Transactions on Programming Languages and Systems*, 15(4):681–705, 1993.

[7] D. Gračanin, P. Srinivasan, and K. P. Valavanis. Parameterized Petri nets and their application to planning and coordination in intelligent systems. *IEEE Transactions on Systems, Man, and Cybernetics*, 24(10):1483–1497, Oct. 1994.

[8] D. Gračanin, L. H. Singh, S. A. Bohner, and M. G. Hinchey. Model-driven architecture for agent based systems. In M. G. Hinchey, J. Rash, W. Truszkowski, and C. Rouff, editors, *Proceedings of the Third NASA Workshop on Formal Approaches to Agent-Based Systems (FAABS III)*, volume 3228 of *Lecture Notes in Computer Science*, Greenbelt, Maryland, 26–28 Apr. 2004. Springer Verlag.

[9] A. Helsinger, K. Kleinmann, and M. Brinn. A framework to control emergent survivability of multi agent systems. In *AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 28–35. IEEE Computer Society, 2004.

[10] A. Helsinger, R. Lazarus, W. Wright, and J. Zinky. Tools and techniques for performance measurement of large distributed multiagent systems. In *AAMAS '03: Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 843–850. ACM Press, 2003.

[11] M. G. Hinchey and S. A. Jarvis. *Concurrent Systems: Formal Development in CSP*. The McGraw-Hill International Series in Software Engineering. McGraw-Hill Book Company, London, 1995.

[12] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International Series in Computer Science. Prentice/Hall International, Englewood Cliffs, New Jersey, 1985.

[13] S. T. Huang. A distributed deadlock detection algorithm for csp-like communication. *ACM Transactions on Programmin Languages and Systems*, 12(1):102–122, 1990.

[14] K. Kleinmann, R. Lazarus, and R. Tomlinson. An infrastructure for adaptive control of multi-agent systems. In *Proceeding of the International Conference on Integration of Knowledge Intensive Multi-Agent Systems*, pages 230–236, 30 Sept.–4 Oct. 2003.

[15] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley, Boston, 2003.

[16] J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, Englewood Cliffs, New Jersey 07632, 1981.

[17] H. Rueß, H. Pfeifer, and F. von Henke. Formalization and reasoning in a reflective architecture. In M. H. Ibrahim, editor, *IJCAI'95 Workshop — Reflection and Metalevel Architectures and their Applications in AI*, Montreal, Canada, July 1995.

[18] S. Schneider. *Concurrent and Real-Time Systems: The CSP Approach*. Worldwide Series in Computer Science. John Wiley & Sons, Ltd, Chichester, 2000.

[19] L. Shan and H. Zhu. Modelling and specifying scenarios and agent behaviour. In *Proceedings of the IEEE/WIC International Conference on Intelligent Agent Technology*, pages 32–38, 13–16 Oct. 2003.

[20] O. Shehory and A. Sturm. Evaluation of modeling techniques for agent-based systems. In *AGENTS '01: Proceedings of the Fifth International Conference on Autonomous Agents*, pages 624–631. ACM Press, 2001.

[21] M. Thome. Multi-tier communication abstractions for distributed multi-agent systems. In *Proceeding of the International Conference on Integration of Knowledge Intensive Multi-Agent Systems*, pages 209–214, 30 Sept.–4 Oct. 2003.

[22] P. H. Welch and J. M. R. Martin. A CSP model for Java multithreading. In *Proceedings of International Symposium on the Software Engineering for Parallel and Distributed Systems*, pages 114–122, 10–11 June 2000.

[23] C. Whitby-Strevens. Transputers—past, present and future. *IEEE Micro*, 10(6):16–19:76–82, Dec. 1990.

[24] D. Xu, R. Volz, T. Ioerger, and J. Yen. Modeling and verifying multi-agent behaviors using predicate/transition nets. In *SEKE '02: Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, pages 193–200. ACM Press, 2002.