Int. J. Appl. Math. Comput. Sci., 2011, Vol. 21, No. 1, 9–25
DOI: 10.2478/v10006-011-0001-6

amcs

FSP AND FLTL FRAMEWORK FOR SPECIFICATION AND VERIFICATION OF MIDDLE–AGENTS

AMELIA BĂDICĂ *, COSTIN BĂDICĂ **

* Business Information Systems Department
University of Craiova, 13 A.I. Cuza St, 200530, Craiova, Romania
e-mail: ameliabd@yahoo.com

**Software Engineering Department
University of Craiova, 107 Decebal Blvd, 200440, Craiova, Romania
e-mail: costin.badica@software.ucv.ro

Agents are a useful abstraction frequently employed as a basic building block in modeling service, information and resource sharing in global environments. The connecting of requester with provider agents requires the use of specialized agents known as middle-agents. In this paper, we propose a formal framework intended to precisely characterize types of middle-agents with a special focus on matchmakers, brokers and front-agents by formally modeling their interactions with requesters and providers. Our approach is based on capturing interaction protocols between requesters, providers and middle-agents as finite state processes represented using FSP process algebra. The resulting specifications are formally verifiable using FLTL temporal logic. The main results of this work include (i) precise specification of interaction protocols depending on the type of middle-agent (this can also be a basis for characterizing types of middle-agents), (ii) improvement of communication between designers and developers and facilitation of formal verification of agent systems, (iii) guided design and implementation of agent-based software systems that incorporate middle-agents.

Keywords: multi-agent system, interaction protocol, process algebra, formal specification.

1. Introduction

Sharing information, resources and services on a global scale is a key function of the modern information society. Advanced digital technologies including software agents, grids and Web services have recently been proposed to support this function. In this context, the ability of parties involved (humans, businesses or software) to flexibly discover each other and to dynamically engage in relationships is very important.

Agents are a useful abstraction frequently employed as a basic building block in modeling service, information and resource sharing in global environments. Typically such an environment contains *requester and provider agents* that are dynamically created and destroyed and consequently do not know each other in advance. Connecting requester agents with provider agents is a crucial issue known as the *connection problem* (Decker *et al*., 1997). Its solution requires the use of additional specialized agents known as *middle-agents* (Decker *et al*., 1997; Yarom *et al*., 2003; Klusch and Sycara, 2001). The activity carried out by middle-agents is called *intermediation*, and it assumes suitable interactions of middle-agents with requesters and providers.

The main contribution of this paper is precise characterization of types of middle-agents—matchmakers, brokers and front-agents—by formally modeling their interactions with requesters and providers in global environments. The paper employs a formal framework for the modeling and analysis of interaction protocols between requesters, providers and middle-agents. In this framework we focus on interactions, i.e., sequences of messages exchanged between parties involved—requesters, providers and middle-agents. In our opinion the results of this work have the following benefits: (i) precise specification of the interactions depending on the type of middle-agent (this can also be a basis for characterizing types of middle-agents); (ii) understanding the requirements of parties involved in such interactions; (iii) improvement of communication between designers and developers and facilitation of formal verification; (iv) guided design and implemen-

tation of software systems that incorporate middle-agents.

Despite the early work of Decker *et al*. (1997), which advocates systematic classification of middle-agents, we noticed that "notions of middle-agents, matchmakers, brokers [...] are used freely in the literature [...] without necessarily being clearly defined" (Klusch and Sycara, 2001). Moreover, most of the references to middle-agents focus in more or less detail on various applications and systems that use different types of middle-agents, highlighting implementation and/or performance issues. Therefore, in this paper, we focus on improving this state-of-affair by presenting and discussing formal models of middle-agents with the goal of highlighting their similarities and differences.

Typical use of middle-agents is encountered in e-commerce applications (Bădică *et al*., 2007; Fasli, 2007; Yarom *et al*., 2003). For example, the model agent-based e-commerce system discussed by Bădică *et al*. (2007) uses middle-agents to connect user buyers on the purchasing side with shops on the selling side in a distributed marketplace. Each user buyer is represented by a *Client* agent, and each shop is represented by a *Shop* agent. The user buyer submits an order to the system for purchasing a product via his or her *Client* agent. The *Client* agent acts as a *Front-agent* (see Section 5) with respect to connecting the user buyer with an appropriate *Shop* agent that provides the requested product. Moreover, the *Client* agent uses a special agent called the *Client Information Center* (*CIC*) that is responsible for providing information which shop in the system sells which products. Thus, it can be easily noticed that the *CIC* is in fact a *Matchmaker* (see Section 5) with respect to connecting the *Client* agent with an appropriate *Shop* agent.

Matchmakers are also available in general-purpose agent platforms. For example, the *Directory Facilitator* (*DF*) is an optional component of an FIPA-compliant agent platform[1] that is present in agent platforms like JADE[2]. The *DF* provides a yellow pages service allowing agents to publicize, unregister and update descriptions of services that are made available for use to other agents in the system.

Formalizing software architectures, components, agents and services has generated a lot of interest during the last years (Zhang *et al*., 2010). Many approaches utilize process algebras as the foundational formalism (Bergstra *et al*., 2001). We follow the trend by grounding our framework for the modeling and analysis of interactions between requesters, providers and middle-agent on *finite state process (FSP) algebra* (Magee and Kramer, 2006). The approach is applied to middle-agent types proposed by Decker *et al*. (1997). Then we show how qualitative properties of their models can be formally defined and verified using *fluent linear temporal logic* (FLTL) (Magee and Kramer, 2006). Our work can be described as an FSP and FLTL-based framework for formal specification and verification of systems with middle-agents. Note that, concerning the specification of software systems, to the best of our knowledge FSP has only been used for the specification of software architectures, processes, components, services, and aspects (Bădică *et al*., 2003; Foster *et al*., 2006; Hennicker and Ludwig, 2005; Xu *et al*., 2009), and it is only recently that FSP has been applied to model software agent interactions (Bădică and Bădică, 2008c).

Our proposal for using FSP and FLTL for this task is motivated by the following: (i) FSP and FLTL are formal specification and verification languages that have a rigorous mathematical foundation; (ii) FSP has a clear and concise syntax and compositional semantics, thus being very appropriate to concisely describe the behavior and interactions in systems with middle-agents; (iii) the definition of properties in FLTL can benefit from earlier work in verification patterns (Dwyer *et al*., 1999); (iv) FSP and FLTL are also well supported by the *Labeled Transition System Analyzer* (LTSA) software tool (Magee and Kramer, 2006); (v) FSP has operational semantics, so it follows that the resulting specifications are executable (can be traced with the help of LTSA); moreover, FSP specifications can be used as a basis for design and software implementation of agent behaviors, for example using finite state machines (Goh *et al*., 2007); (vi) recently, FSP has been extended to *Modal Transition Systems* formalism (MSP) for modeling and analysis in the presence of partial information about system behavior; MSP is currently supported by the *Modal Transition System Analyzer* tool (MTSA); MSP and MTSA provide better support in the context of current software development practices (D'Ippolito *et al*., 2008).

We start in Section 2 with a literature overview of related works on middle-agents and process-algebraic approaches to the formalization of agent systems. In Section 3 we state the basic assumptions underlying our modeling. In Section 4 we introduce FSP and the guidelines of modeling agent interactions with FSP. Then, in Section 5, we present detailed FSP models of *Matchmaker*, *Front-agent*, and *Broker* middle-agents. In Section 6 we show how FLTL can be used to formally check the developed models against qualitative properties. We follow in Section 7 with experimental evaluation and verification of the models. The final part of the paper contains our conclusions.

2. Related works

Research literature referring to middle-agents is quite rich: a search on Google for the words "middle-agents" returns about 20000 references. Despite this relatively large num-

[1] http://www.fipa.org.
[2] http://jade.cselt.it.

ber of references, which indicates that there is a growing interest in the subject of middle-agents, our literature overview indicates that only very few papers address the problem of a concise definition of middle-agents in terms of their interaction capabilities with provider and requester agents, to precisely characterize their typology. Rather, many references to middle-agents focus in more or less detail on various applications and systems that use different types of middle-agents, most often brokers and/or matchmakers (see Klusch and Sycara, 2001), and highlighting implementation and/or performance issues.

The starting point of our work is the classification of middle-agents introduced in the seminal work of Decker *et al*. (1997). Based on privacy assumptions about what is initially known in an interaction by requesters, middle-agents, and providers about requester preferences and provider capabilities, Decker *et al*. (1997) proposed nine types of middle-agents: *Broadcaster*, *Matchmaker*, *Front-agent*, *Anonymizer*, *Broker*, *Recommender*, *Blackboard*, *Introducer*, and *Arbitrator* (see Table 1). Note that in this context the term "initially" means "before the interaction between the requester and the provider".

Wong and Sycara (2000) attempt to propose a more systematic classification of middle-agents by refining the classification introduced by Decker *et al*. (1997). The authors suggested the use of six binary dimensions for characterizing middle-agent types. However, while noting that not all $2^6 = 64$ combinations are meaningful, Wong and Sycara (2000) do not clearly show how to appreciate which are meaningful and which are not, leaving this decision up to the designer's intuition. After reviewing that taxonomy, our conclusion is that, on the one side not all proposed dimensions are relevant for our work, and, on the other, that we do not totally agree with that taxonomy. For example, according to Dimension 1, Wong and Sycara (2000) claim that "providers and requesters should act in a complementary way with respect to who sends information to middle-agents, i.e., if providers push, requesters pull and vice-versa". In our opinion the main characteristic of a *Broker* is that both providers and requesters push, while the *Broker* has the option to choose the most effective matching provider capability for a given request and the most effective matching requester preference for a given provider capability. Obviously, the taxonomy proposed by Wong and Sycara (2000) fails to correctly describe such a behavior.

Alagar and Holliday (2002) suggest the use of labeled transition systems (LTSs) for modeling agent types. However, this paper does not address the formal modeling of middle-agents. Klusch and Sycara (2001) discuss the modeling of matchmakers and brokers using input/output automata (following the initial proposal of Wong and Sycara (2000)), while Hristozova and Sterling (2003) informally describe an application of middle-agents and ontologies in the area of value-added publishing.

Our "agents-as-processes" modeling approach is not entirely new. Esterline *et al*. (2006) propose the use of π-calculus (Milner, 1999) and show models of a prototype agent system called LOGOS for an unattended grounds operation-center using a fault resolution scenario. However, while the authors show how to use a software tool for checking the proposed specifications, they do not provide any experimental results—only general modeling guidelines are given. Rouff *et al*. (2006) also focus on the LOGOS agent system, but using CSP process algebra (Hoare, 1985). While that paper discusses some benefits of the method (detecting race conditions, message omissions, and better understanding of the system), it still does not provide any experimental results and does not discuss the practical problems encountered.

Bădică *et al*. (2003) propose a formal model of business processes captured as role activity diagrams, using FSP. Another work by Bădică *et al*. (2007) introduces a general framework for FSP modeling of middle-agents and shows how this framework can be used to model a *Recommender* agent. Using the same idea, Bădică and Bădică (2008c) present a model of an *Arbitrator* middle-agent for coordination of participants in single-item English auctions. Another work reported by Bădică and Bădică (2008b) is an extension of the research by Bădică *et al*. (2007) as well as Bădică and Bădică (2008c) for modeling *Matchmaker*, *Front-agent* and *Broker* middle-agents, including both theoretical and experimental results. The papers by Bădică and Bădică (2008a; 2008e; 2009a) focus on introducing verification models of systems that contain middle-agents, using FLTL temporal logic. Bădică and Bădică (2008d) show how models of *Matchmaker* (Bădică and Bădică, 2008b) and *Arbitrator* for English auctions (Bădică and Bădică, 2008c) can be combined into a more complex system for agent-based auctions with matchmaking capabilities. Moreover, in another work (Bădică and Bădică, 2009b), we extend the auction model of Bădică and Bădică (2008c) to formally specify and verify the agent-based auction service discussed by Dobriceanu *et al*. (2009) using FSP and FLTL. Finally, in a recent paper (Bădică *et al*., 2009), we show how our formal framework can be applied to model protocol-based service coordination in agent systems. Actually, this coordination approach is a natural extension of the service model introduced by Bădică and Bădică (2009b).

A limitation of our process-algebraic method that we would like to address in the future is that it does not provide features for modeling intelligence aspects of middle-agents, i.e., their decision making, strategic and reasoning capabilities. In our opinion this would require the extension of the models with abilities for representing state variables, utilities and knowledge. Some related works in this area include those by Merayo *et al*. (2007) as well as Miller and McBurney (2007). Merayo *et al*. (2007) introduce an extension of finite state machines—*extended*

Table 1. Middle-agent types.

Preferences/Capabilities	Provider only	Provider & MA	All
Requester only	Broadcaster	Front-agent	Matchmaker
Requester & MA	Anonymizer	Broker	Recommender
All	Blackboard	Introducer	Arbitrator

utility state machines (EUSMs) for modeling strategic aspects of intelligent agents. The EUSM supports a richer representation of agents behavior as compared to the FSP approach by enhancing their state representation with state variables and utility functions. Miller and McBurney (2007) propose a process algebraic approach enhanced with constraint-reasoning capabilities—$\mathcal{RASA}$ for the specification of semantic aspects of agent interaction protocols. Using $\mathcal{RASA}$, intelligent agents can reason at run-time about their possible actions and effects in order to tune their performances. However, Miller and McBurney (2007) only present a simplistic example and it is not clear when and how the $\mathcal{RASA}$ framework should be applied in practice. Finally, Rahimi *et al*. (2002) propose an extension of higher-order π-calculus to represent the intelligence component of an autonomous agent.

As for as their application areas are concerned, middle-agents were recently put forward by Fasli (2007) in the context of e-commerce. The author covers in some detail *Matchmaker*, *Broker*, *Broadcaster* and *Recommender* with a focus on interaction protocols and languages for describing requester preferences and provider capabilities. However, while the *Matchmaker* description fits within the models that we propose in our paper, note that *Broker* described by Fasli (2007) actually corresponds to our model of *Front-agent*.

Communities of middle-agents were also used for self-organization of distributed information systems according to dynamically changing user preferences and characteristics (Wang, 2002). The focus of this work was set on flexibility and scalability issues, rather than formal modeling and understanding of middle-agent typology.

3. Modeling assumptions

In this section we introduce the basic modeling assumptions that we employ in our modeling and analysis of interactions between requesters, providers and middle-agents. These assumptions include (i) abstracting away from domain and language specific details, and (ii) taking into account the distinction between subscriptions and ordinary requests.

3.1. Domain and language specific details. There is certainly a domain dependent component of intermediation, involving domain or language specific aspects of preferences and capabilities of requesters and providers.

For example, there are clearly domain-specific differences between searching and intermediation of business or holidays travel packages in an e-travel environment and discovering learning resources (for example, a programming course) for an e-learning application. However, we would expect that interactions necessary to search for a travel package that has a given set of characteristics by querying travel agencies and to search for a programming course whose completion would guarantee a certain level of competence in a given programming language are basically the same.

Following this idea, in our framework we abstract away from (i) the specific and domain dependent details of the environment, i.e., we do not care if the target of our analysis is, for example, related to learning, commerce or travel, and (ii) actual languages used for representing requests, responses, preferences, and capabilities, i.e., we do not care if requests and responses are for example, represented using FIPA ACL[3] and capabilities and preferences using FIPA SL[4]. In other words, we abstract away from the matching activity and, instead, we focus on the dynamics of interactions, i.e., sequences of messages exchanged between involved parties—requesters, providers and middle-agents.

One apparent weakness introduced by adopting this assumption is setting the focus on control modeling details (agents and their interactions), rather than on data modeling details. This claim is supported by the observation that the goal of real world implementations of agent systems with middle-agents is usually mediation and data transfer. While abstraction from data type specification can be of course limiting for the achievement of the goal of precise specification and design of an agent system as part of a software development process, this abstraction can be extremely useful for characterizing types of middle-agents and understanding the interaction requirements of parties involved. Moreover, if we also consider the goal of formal verification of agent interactions, data abstraction can be extremely useful for reducing the state-space explosion in model checking (Zhang *et al*., 2010). Finally, note that there are proposals to enhance process algebraic approaches with the ability of handling data by means of equational abstract data types, like μCRL, de-

[3] http://www.fipa.org/specs/fipa00061.
[4] http://www.fipa.org/specs/fipa00008.

scribed in detail by Fokkink (2007). However, this approach is outside the scope of this paper and therefore we will not pursue it here.

3.2. Subscriptions versus ordinary requests. Following a recent work of Mbala *et al*. (2006), we found it useful to distinguish from the very beginning between *subscriptions* and ordinary requests. Subscriptions can be defined as long-lived or *reproducible* "requests" (Mbala *et al*., 2006), while ordinary requests are *non-reproducible* (simply called requests in what follows). The difference is that a successful subscription will be continually served until it is eventually canceled (by the requester—subscriber in this particular case), while an ordinary request is issued and served at most once or eventually abandoned (by the middle-agent). For example, a buyer agent can submit a request to a directory agent that manages information about seller agents and the categories of products they sell to find out a list of potential sellers of a given product. Once issued, the request is served by providing the list of sellers and the conversation between the buyer agent and the directory agent ends. Differently, a travel agent can subscribe to a weather information provider that periodically sends weather information to all her subscribers.

Note that the distinction between requests and subscriptions is related to serving requests either in *pull* or in *push* mode (Wong and Sycara, 2000). A request served in pull mode is either immediately served or declined by the middle-agent, while a request served in push mode is memorized and the requester notified later if the request can be served or it is abandoned by the middle-agent.

The situation is symmetric from the point of view of providers. In this case, a provision (of services, resources or information) can be either *non-reproducible* or *reproducible*. In the first case, the provision, once consumed, is no longer available until explicitly made so by the provider, while in the second case the provision is continuously available until explicitly canceled by the provider. An example of a non-reproducible provision is a seller that sells a book on eBay—once the book is sold, it will no longer be available for sale. An example of a reproducible provision is a weather information provider like Weather Underground.

Note that reproducibility and non-reproducibility of requests and provisions will affect the interaction patterns between requesters, providers and middle-agents. As the focus of our work is to precisely characterize middle-agents types as proposed by Decker *et al*. (1997), we had to assume that requests are *non-reproducible* (i.e., we focus on requests, and not subscriptions) and that provisions are *reproducible*. However, the other situations are also conceivable, leading obviously to different models.

4. Modeling framework

Before presenting FSP models of middle-agents we first briefly review FSP and then we introduce the guidelines of our modeling approach.

4.1. Overview of FSP. FSP is an algebraic specification technique of concurrent and cooperating computational processes as finite state labeled transition systems (LTSs hereafter). FSP allows a more compact and easy to manage description of an LTS, rather than by directly describing it as a list of states and transitions between states.

An FSP model consists of a finite set of sequential and/or composite process definitions. Additionally, a sequential process definition consists of a sequence of one or more definitions of local processes. A process definition consists of a process name associated with a process term.

FSP uses a rich set of constructs for process terms (see Magee and Kramer, 2006) . For the purpose of this paper we are using the following constructs: (i) action prefix ($a \rightarrow P$), nondeterministic choice ($P|Q$), and process alphabet extension ($P + \{a_1, \ldots, a_n\}$) for sequential process terms, and (ii) parallel composition ($P||Q$) and relabeling ($P/\{new_1/old_1, \ldots, new_k/old_k\}$) for composite process terms.

FSP has operational semantics given via an LTS. The mapping of an FSP term to an LTS is described in detail by Magee and Kramer (2006), and it follows the intuitive meaning of FSP constructs.

4.2. Modeling guidelines. We propose the following guidelines to be applied in the development of FSP models of middle-agents:

(i) Agents are modeled as FSP processes. As our process language is FSP, we chose to model agents types as sequential processes. Instantiation of an agent of a given type can be defined by invoking the associated process with a suitable relabeling of its alphabet.

(ii) A multi-agent system is modeled as a parallel composition of processes. It follows that communication between agents is modeled by appropriately utilizing the synchronization capabilities of FSP. In FSP, synchronization of processes of a parallel composition is done by default on their common alphabets. Accordingly, special care should be taken in order to accurately model agent communication using FSP synchronization. Depending on circumstances this will require relabeling and/or alphabet extension of local processes (see, for example, the *Matchmaker* model from Fig. 1). Relabeling is utilized when a new process is instantiated via its process name, so basically it is a reusability mechanism. Alphabet extension is utilized to correctly model parameter passing between processes via action synchronization. This modeling is not so obvious,

and therefore we explain it by means of an example. Let us suppose that a process Q must pass a parameter v with the value in a finite set $\mathcal{V}$ to a process P. The decision of what value is passed by Q to P is taken by Q and usually only some of the values (not all) in $\mathcal{V}$ can be passed—let us denote this set of values by $\mathcal{V}' \subseteq \mathcal{V}$. This situation is modeled by a set of actions $send(v)$ indexed with all the values of v that can be set by Q (i.e., values from the set $\mathcal{V}'$). Actions $send(v)$ must be performed by both P and Q, but because P does not know the value of v that is passed by Q, it must be able to execute all the actions $send(v \in \mathcal{V})$, so all of them will be automatically included in its alphabet. As Q will execute an action $send(v)$ such that v is a value that must be passed to P, it follows that only actions $send(v \in \mathcal{V}')$ will be part of Q's alphabet. Communication between P and Q is modeled by considering the parallel composition of P and Q. Note that if $v \in \mathcal{V}'$ then action synchronization will correctly model passing of v from Q to P. However, if $v \in \mathcal{V} \setminus \mathcal{V}'$, the process P will be able to execute $send(v)$, while Q will not, so execution of $P \parallel Q$ will proceed without synchronization with action $send(v)$. This incorrect behavior can be corrected by extending the alphabet of Q, so instead of Q we shall use $Q' = Q + \{send(v \in \mathcal{V})\}$. In this situation, if $v \in \mathcal{V} \setminus \mathcal{V}'$, then the processes P and Q' must synchronize on their common alphabets, and independent execution of P without synchronization with Q' will not be possible for actions $send(v)$ where $v \in \mathcal{V} \setminus \mathcal{V}'$.

Note that alphabet extension is utilized in the *Matchmaker* model (see Section 5.1).

(iii) We assume that $\mathcal{R}$ is the set of requesters and $\mathcal{P}$ is the set of providers. The identity of requester and provider agents is explicitly represented by integer indexes $r \in \mathcal{R}$ and $p \in \mathcal{P}$.

(iv) Requests made by requester agents are indexed with the requester ID. Requests made to the providers are indexed with the provider ID. It follows that (a) requests from requesters to middle-agents are indexed only with the ID of the requester—action $request(r)$, and (b) requests from middle-agents to providers are indexed only with the ID of the providers; (c) requests made by requesters directly to providers are indexed with both the IDs of the requester and the provider—action $request\_to\_provider(r, p)$. Note that details like request preferences and service capabilities are abstracted away from our models, as we consider them unnecessary to understand the specific particularities of interaction for each type of middle-agent. Accordingly, this assumption means that details of a request/capability are "incorporated" in the ID of the requester/provider.

(v) The matching operation is modeled as a relation $\mathcal{M}$ between the sets $\mathcal{P}$ of providers and $\mathcal{R}$ of requesters, i.e., $\mathcal{M} \subseteq \mathcal{P} \times \mathcal{R}$. If r is a requester ID then the set P of IDs of matching providers that results from a matching operation is $P = \mathcal{M}(r)$. Symmetrically, if p is a provider ID then the set R of the IDs of matching requesters that results from a matching operation is $R = \mathcal{M}^{-1}(p)$.

(vi) We define some action naming conventions with the role of standardizing the communication interfaces of the agents with the exterior environment consisting of requesters, providers, and possibly other middle-agents. A provider will use the action *offer* to register a capability and action *withdraw* to unregister a capability with the middle-agent. As for the provider's service interface, we use the convention *receive_request* / *send_reply* to model the receipt of a service request and return of a reply. A requester will use the action *request* for requesting a service from the middle-agent. If the service is not intermediated by the middle-agent, then the middle-agent will respond with the action *tell*. If the service is intermediated by the middle-agent, then the middle-agent will respond either with the action *success* if the serve provision was successful or with the action *fail* in the case of failure to service the requester.

5. FSP models of *Matchmaker*, *Front-agent* and *Broker*

Based on our literature overview, we noticed that frequently utilized middle-agents for connecting provider and requester agents are *Matchmaker*, *Front-agent*, and *Broker*. In this section, we present and discuss their FSP models, using the framework and the modeling assumptions introduced in the previous sections. Taking into account that the modeling principles are general and the resulting models are complex enough, one can easily apply this modeling approach to the other types of middle-agents (see, e.g., the model of *Recommender* discussed by Bădică *et al*. (2007)).

Note that we start with an intuitive description of types of middle-agents that aims at providing a concise natural language description of middle-agent interaction patterns with requesters and providers, emphasizing the initial assumptions that were used for their classification (Decker *et al*., 1997) and follow with the specification and analysis of their formal FSP models.

5.1. Matchmaker middle-agent in FSP. A *Matchmaker* middle-agent (also known as *Yellow-pages*, see Table 1) assumes that, before the interaction between the requester and the provider, requester preferences are known only to the requester, while provider capabilities are or will become known to all interaction participants. This means that a provider will have to advertise its capabilities with *Matchmaker*, and *Matchmaker* is responsible for matching a request with registered capabilities advertisements. However, the fact that provider capabilities are initially known also by the requester means that the result of the matching (i.e., the set of matching providers) will be

returned by *Matchmaker* to the requester (provider capabilities become thus known to the requester), and the choice of the matching provider is the responsibility of the requester. Consequently, the transaction is not intermediated by *Matchmaker*, as would be the case, for example, with *Broker* or *Front-agent*.

The block diagram and FSP specification of a system with requesters, providers and *Matchmaker* are shown in Fig. 1.

The *Provider* agent registers its capability offer (the action $offer$) with *Matchmaker* and then enters a loop where it receives requests from *Requester* agents via the action $receive\_request$ and processes and replies accordingly via the action $send\_reply$. Note that *Provider* can also withdraw a registered capability offer, and while its capability is not registered it always refuses to serve a request (the action $refuse\_request$).

The *Requester* agent submits a request to *Matchmaker* (the action $send\_request$) and then waits for a reply. *Matchmaker* replies with a set of matching providers (the action $tell$ with the argument $\mathcal{M}(r) \cap P$ representing the set of matches; here P is the set of registered providers and $\mathcal{M}(r)$ is the set of matching providers). Then *Requester* has the option to choose which provider from set P to contact for performing the service (the action $send\_request\_to\_provider$ with argument $p \in P$ representing the chosen provider). Finally, *Requester* waits for a reply from the contacted provider (the action $receive\_reply$).

The *Matchmaker* agent registers and unregisters *Provider* offers and answers *Requester* requests for matching offers. *Matchmaker* informs *Requester* about available registered offers (the action $tell$). Note that *Requester* is responsible for choosing an appropriate matching offer from the available matching offers (the action $send\_request\_to\_provider$). This complicates a bit the behavior of *Requester* compared with *Front-agent* and *Broker* cases (see Subsections 5.2 and 5.3).

Special care is taken to accurately model agent communication using FSP synchronization. The *Matchmaker* model requires alphabet extension (construct $\{tell(r' \in R, P' \subseteq \mathcal{P})\}$ in Fig. 1) to correctly model communication between *Matchmaker* and *Requester*.

A critical situation may occur when a matching offer is found but the matching *Provider* chooses to cancel its offer by unregistering it with *Matchmaker* before it is actually contacted by *Requester*. This ability of *Provider* is modeled with the action $refuse\_request$. Note that this situation cannot occur with *Front-agent* and *Broker* (remember that both *Front-agent* and *Broker* intermediate the request on behalf of *Requester*), so we did not have to model this ability of *Provider* in those cases.

5.2. Front-agent middle-agent in FSP. A *Front-agent* middle-agent (also known as *Proxy*, see Table 1) assumes that, before the interaction between the requester and the provider, requester preferences are known only to the requester, while provider capabilities are known both to provider and the middle-agent. This means that the provider will have to advertise its capabilities with *Front-agent*, and *Front-agent* is responsible for matching a request with registered capabilities advertisements. Additionally, as the provider capabilities will not be known to the requester, *Front-agent* also has the responsibility of intermediating the transaction between the requester and the matching provider (this is why often this type of middle-agent is called *Broker* rather than *Front-agent*; in our opinion a true *Broker*, is different, see below).

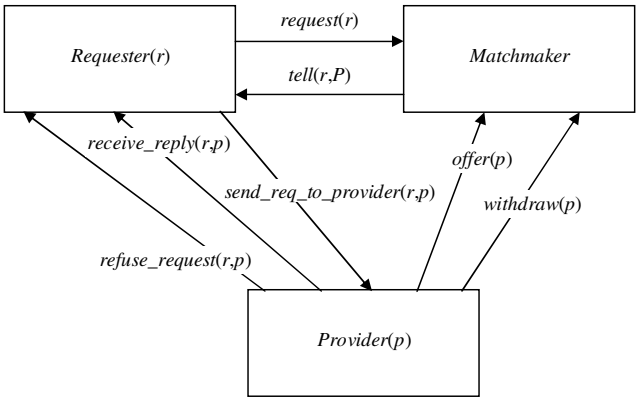
The block diagram and FSP specification of a system composed of requesters, providers and *Front-agent* are shown in Fig. 2.

The *Provider* agent is similar to the *Matchmaker* case. The *Requester* agent is simpler than the *Matchmaker* case: it submits a request to *Front-agent* (the action $send\_request$) and then waits for *Front-agent* to either resolve that request (the action $success$) or fail (the action $fail$).

The *Front-agent* agent processes requests from *Requester* agents and registers offers from *Provider* agents. Note that, unlike *Matchmaker*, *Front-agent* has the responsibility of choosing an appropriate matching provider from the available matching providers $\mathcal{M}(r) \cap P$ (here P is the set of registered providers and $\mathcal{M}(r)$ is the set of matching providers) using the action $fragent\_request$ and to resolve the request (the action $fragent\_request$). Finally, the result is passed to the *Requester* agent using the action $success$. In conclusion, the actual *Provider* that fulfils the request for *Requester* on behalf of *Front-agent* is hidden from *Requester* that issued the request.

5.3. Broker middle-agent in FSP. A *Broker* middle-agent (see Table 1) assumes that, before the interaction between the requester and the provider, requester preferences are known only to the requester and the middle-agent and provider capabilities are known only to the provider and the middle-agent. The crucial point is, however, that requester preferences will not be known to the provider and provider capabilities will not be known to the requester. This means that *Broker* will truly intermediate transactions between providers and requesters in *both* directions: (i) if a requester submits a request either it cannot be matched and it is registered with *Broker* or it is matched with a provider capability and then transaction is intermediated by *Broker*, and (ii) if a provider advertises a capability then the capability is registered with *Broker* and it is also matched against registered requests; neither match is found and nothing more happens or matches are found and corresponding transactions are intermediated by *Broker*.

The block diagram and FSP specification of a system composed of requesters, providers and *Broker* are shown



$$
\begin{array}{lcl}
Provider & = & (offer \rightarrow ProcessRequest \mid \\
 & & receive\_request \rightarrow refuse\_request \rightarrow Provider), \\
ProcessRequest & = & (receive\_request \rightarrow send\_reply \rightarrow ProcessRequest \mid \\
 & & withdraw \rightarrow Provider).
\end{array}
$$

$$
\begin{array}{lcl}
Requester & = & (send\_request \rightarrow WaitReply), \\
WaitReply & = & (tell(P \subseteq \mathcal{P}) \rightarrow \textbf{if } P \neq \emptyset \textbf{ then } ContactProvider(P) \textbf{ else } Requester), \\
ContactProvider(P \subseteq \mathcal{P}) & = & (\textbf{while } P \neq \emptyset \; send\_request\_to\_provider(p \in P) \rightarrow \\
 & & \{receive\_reply(p), refuse\_request(p)\} \rightarrow Requester).
\end{array}
$$

$$
\begin{array}{lcl}
Matchmaker & = & Matchmaker(\emptyset), \\
Matchmaker(P \subseteq \mathcal{P}) & = & (request(r \in \mathcal{R}) \rightarrow MatchReq(r, P) \mid \\
 & & offer(p \in \mathcal{P} \setminus P) \rightarrow Matchmaker(P \cup \{p\}) \mid \\
 & & withdraw(p \in P) \rightarrow Matchmaker(P \setminus \{p\})), \\
MatchReq(r \in \mathcal{R}, P \subseteq \mathcal{P}) & = & (tell(r, \mathcal{M}(r) \cap P) \rightarrow Matchmaker(P)) + \{tell(r' \in \mathcal{R}, P' \subseteq \mathcal{P})\}.
\end{array}
$$

$$
\begin{array}{lcl}
Requester(r \in \mathcal{R}) & = & Requester/\{request(r)/send\_request, tell(r, P \subseteq \mathcal{P})/tell(P), \\
 & & send\_request\_to\_provider(r, p \in \mathcal{P})/send\_request\_to\_provider(p), \\
 & & receive\_reply(r, p \in \mathcal{P})/receive\_reply(p), refuse\_request(r, p \in \mathcal{P})/refuse\_request(p)\}. \\
Provider(p \in \mathcal{P}) & = & Provider/\{offer(p)/offer, \\
 & & send\_request\_to\_provider(r \in \mathcal{R}, p)/receive\_request, \\
 & & receive\_reply(r \in \mathcal{R}, p)/send\_reply, refuse\_reply(r \in \mathcal{R}, p)/refuse\_reply\}. \\
System & = & Matchmaker \mid\mid (\mid\mid_{r \in \mathcal{R}} Requester(r)) \mid\mid (\mid\mid_{p \in \mathcal{P}} Provider(p)).
\end{array}
$$

Fig. 1. System with the *Matchmaker* middle-agent.

in Fig. 3. *Provider* and *Requester* agents are similar to the *Front-agent* case.

Broker processes requests from *Requesters* and processes and registers offers from *Providers*. If a request can be served based on available matching offers, then *Broker* behaves similarly to *Front-agent*. Unlike *Front-agent*, if a request cannot be served based on currently registered offers, then, rather than reporting failure, the request is recorded until either (i) a new matching offer is registered with the *Broker*, or (ii) the request is deemed failed. Note that when a new offer is registered, *Broker* determines the set of recorded (i.e., not yet served) matching requests – $\mathcal{M}^{-1}(p) \cap R$ (note that $\mathcal{M}^{-1}(p)$ is the set of matching requesters and R is the set of recorded requesters) and serves them using the matching provider p – $ContactProviderOff$ sub-process in Fig. 3.

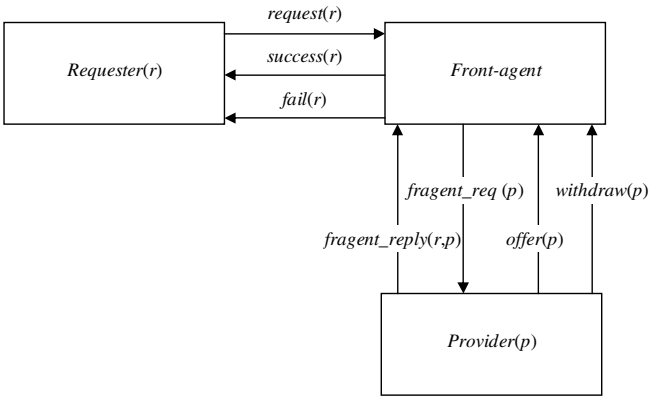
5.4. System properties. A basic desirable property of systems with middle-agents is that they are free of deadlocks. The result is formally stated as follows.

Proposition 1. *The systems with* Matchmaker*,* Front-agent *and* Broker *shown in Figs. 1–3 are deadlock free.*

Proof. We consider only the proof for the *Matchmaker* system. The proofs for *Front-agent* and *Broker* follow the pattern and are therefore omitted.

Let us consider an arbitrary system state S. As the system is a parallel composition of processes, state S is composed of sub-states corresponding to *Matchmaker* and to each of the *Provider* and *Requester* processes. Any progress from S will be caused by an interaction between two processes: *Matchmaker* with *Provider*, *Matchmaker* with *Requester* or *Requester* with *Provider*. Note that *Matchmaker* can be in one of two states: (i) *Matchmaker(*P*)* with $P \subseteq \mathcal{P}$; (ii) $MatchReq(r, P)$ with $r \in \mathcal{R}$ and $P \subseteq \mathcal{P}$.

In the first case, this means that *Matchmaker* finalized to process a request and is waiting for a new one. If a new request is available, we are done. If *Provider* is available to register/unregister a capability offer, we are again done. Otherwise, this means that all requesters submitted requests to providers and wait for service. In this case, we randomly pick a pair $Requester(r)$ and $Provider(p)$ with $r \in \mathcal{R}$ and $p \in \mathcal{P}$, and the system will proceed with interaction between $Requester(r)$ and $Provider(p)$.



$$
\begin{array}{lcl}
Provider & = & (offer \rightarrow ProcessRequest),\\
ProcessRequest & = & (receive\_request \rightarrow send\_reply \rightarrow ProcessRequest \mid\\
 & & withdraw \rightarrow Provider).\\
 & & \\
Requester & = & (send\_request \rightarrow WaitReply),\\
WaitReply & = & (\{sucess, fail\} \rightarrow Requester).\\
 & & \\
Frontagent & = & Frontagent(\emptyset),\\
Frontagent(P \subseteq \mathcal{P}) & = & (request(r \in R) \rightarrow ResolveReq(r, P) \mid\\
 & & offer(p \in \mathcal{P} \backslash P) \rightarrow Frontagent(P \cup \{p\}) \mid\\
 & & withdraw(p \in P) \rightarrow Frontagent(P \setminus \{p\})),\\
ResolveReq(r \in \mathcal{R}, P \subseteq \mathcal{P}) & = & (\textbf{if } \mathcal{M}(r) \cap P = \emptyset \textbf{ then } fail(r) \rightarrow Frontagent(P)\\
 & & \textbf{else } ContactProvider(r, \mathcal{M}(r) \cap P, P)),\\
ContactProvider(r \in \mathcal{R}, P' \subseteq \mathcal{P}, P \subseteq \mathcal{P}) & = & (\textbf{while } P' \neq \emptyset \, fragent\_req(p \in P') \rightarrow fragent\_reply(p) \rightarrow\\
 & & success(r) \rightarrow Frontagent(P)).\\
 & & \\
Requester(r \in \mathcal{R}) & = & Requester/\{request(r)/send\_request, fail(r)/fail, success(r)/success\}.\\
Provider(p \in \mathcal{P}) & = & Provider/\{offer(p)/offer, fragent\_req(p)/receive\_request,\\
 & & fragent\_reply(p)/send\_reply, withdraw(p)/withdraw\}.\\
System & = & Frontagent \,||\, (||_{r \in \mathcal{R}} Requester(r)) \,||\, (||_{p \in \mathcal{P}} Provider(p)).
\end{array}
$$

Fig. 2. System with the *Front-agent* middle-agent.

In the second case, progress will occur through interaction between $Matchmaker$ and $Requester(r)$, as $Requester(r)$ is definitely in state $WaitReply$. ■

6. FLTL modeling of system properties

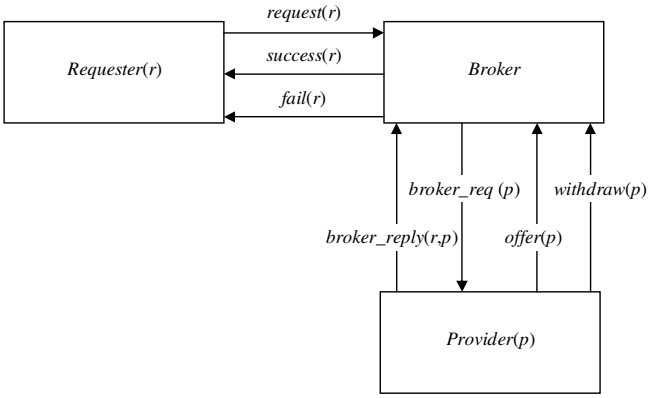
In this section we show how qualitative properties can be defined for systems with requesters, providers and middle-agents. Clearly, there are many types of properties that can be defined for multi-agent systems, focusing either on typical features of agents like autonomy, proactiveness, reactiveness and adaptivity (Brazier *et al*., 2004), or on more specific aspects like those related to complex sequences of messages that agents may exchange during a negotiation process (Podorozhny *et al*., 2007). A full coverage of types of properties that are specific to each type of middle-agent would require a more elaborated treatment and it is therefore beyond the scope of this paper. Nevertheless, in this section we exploit the FSP models introduced in the previous section by showing (using examples) how prototypical qualitative properties can be defined in temporal logic for systems with *Matchmaker* and *Front-agent* middle-agents (Bădică and Bădică, 2008a).

Formal modeling of agent systems has the advantage that models can be systematically checked against user-defined properties. A property is defined by a statement that should be true for all the possible execution paths of the system. A property is used to describe a desirable feature of the system behavior. The definition and analysis of properties of middle-agents have the advantage that they enable a formal and concise, rather than informal and speculative, comparison of types of middle-agents.

Properties of software systems are usually expressed as formulas in a temporal logic language (Clarke *et al*., 1986). Temporal logics are a subclass of modal logics specially tailored for declarative specification of properties of dynamic systems defined as labeled transition systems. Clearly, multi-agent systems that contain middle-agents are a special class of software systems composed of many agents that dynamically interact and coordinate their actions by message exchange.

A property holds if the associated formula is true for all the possible executions of the system, as it is described by the system model. For system models captured using FSP it has been shown that a very convenient temporal logic for property expression is *fluent linear temporal logic* (FLTL) (Magee and Kramer, 2006).

In FLTL, primitive properties are expressed using *fluents*. A fluent is a property whose truth is triggered by an



$$
\begin{array}{lcl}
Provider & = & (offer \rightarrow ProcessRequest), \\
ProcessRequest & = & (receive\_request \rightarrow send\_reply \rightarrow ProcessRequest \mid \\
 & & withdraw \rightarrow Provider). \\
 & & \\
Requester & = & (send\_request \rightarrow WaitReply), \\
WaitReply & = & (\{sucess, fail\} \rightarrow Requester). \\
 & & \\
Broker & = & Broker(\emptyset, \emptyset), \\
Broker(R \subseteq \mathcal{R}, P \subseteq \mathcal{P}) & = & (request(r \in \mathcal{R} \setminus R) \rightarrow ResolveReq(r, R, P) \mid \\
 & & offer(p \in \mathcal{P} \setminus P) \rightarrow ResolveOff(p, R, P) \mid \\
 & & withdraw(p \in P) \rightarrow Broker(R, P \setminus \{p\}) \mid \\
 & & \textbf{while } R \neq \emptyset \; fail(r \in \mathcal{R}) \rightarrow Broker(R \setminus \{r\}, P)), \\
ResolveReq(r \in \mathcal{R}, R \subseteq \mathcal{R}, P \subseteq \mathcal{P}) & = & (\textbf{if } \mathcal{M}(r) \cap P = \emptyset \textbf{ then } Broker(R \cup \{r\}, P) \\
 & & \textbf{else } ContactProviderReq(r, R, \mathcal{M}(r) \cap P, P)), \\
ContactProviderReq(r \in \mathcal{R}, R \subseteq \mathcal{R}, P' \subseteq \mathcal{P}, P \subseteq \mathcal{R}) & = & (\textbf{while } P' \neq \emptyset \; broker\_req(p \in P') \rightarrow broker\_reply(p) \rightarrow success(r) \rightarrow \\
 & & Broker(R, P)), \\
ResolveOff(p \in \mathcal{P}, R \subseteq \mathcal{R}, P \subseteq \mathcal{P}) & = & (\textbf{if } \mathcal{M}^{-1}(p) \cap R = \emptyset \textbf{ then } Broker(R, P \cup \{p\}) \\
 & & \textbf{else } ContactProviderOff(p, \mathcal{M}^{-1}(p) \cap R, R, P)), \\
ContactProviderOff(p \in \mathcal{P}, R' \subseteq \mathcal{R}, R \subseteq \mathcal{R}, P \subseteq \mathcal{P}) & = & (\textbf{if } R' \neq \emptyset \textbf{ then } broker\_req(p) \rightarrow broker\_reply(p) \rightarrow \\
 & & success(r \in R') \rightarrow ContactProviderOff(p, R' \setminus \{r\}, R \setminus \{r\}, P) \\
 & & \textbf{else } Broker(R, P)). \\
 & & \\
Requester(r \in \mathcal{R}) & = & Requester/\{request(r)/send\_request, fail(r)/fail, success(r)/success\}. \\
Provider(p \in \mathcal{P}) & = & Provider/\{offer(p)/offer, fragent\_req(p)/receive\_request, \\
 & & fragent\_reply(p)/send\_reply, withdraw(p)/withdraw\}. \\
System & = & Broker \mid\mid (\mid\mid_{r \in \mathcal{R}} Requester(r)) \mid\mid (\mid\mid_{p \in \mathcal{P}} Provider(p)).
\end{array}
$$

Fig. 3. System with the *Broker* middle-agent.

initiating event and that holds until the signalling of a terminating event. In FSP, it is natural to model initiating and terminating events by executing specific actions. Therefore, following (Magee and Kramer, 2006), a fluent is defined as a triple:

$$\textbf{fluent } F = \langle \{i_1, \ldots, i_m\}, \{t_1, \ldots, t_n\} \rangle \textbf{ initially } B,$$

where (i) $\{i_1, \ldots, i_m\}$ and $\{t_1, \ldots, t_n\}$ are disjoint sets of *initiating events* and *terminating events* and (ii) B is true or false and represents the initial value of F (when the initial value B of the fluent is not given, it is assumed to be false by default). When any of the initiating actions is observed, F becomes true and remains so until any of the terminating actions is observed.

Every action a defines a singleton fluent $F(a)$ having a as the single initiating action and the rest of all actions as terminating actions as follows:

$$\textbf{fluent } F(a) = \langle \{a\}, A \setminus \{a\} \rangle.$$

A singleton fluent $F(a)$ is usually written as a in FLTL formulas.

FLTL formulas are built over fluent propositions (including singleton fluents) using the usual logical operators $\wedge, \vee, \rightarrow, \neg$ and temporal operators $\mathbf{X}$ (next), $\mathbf{U}$ (until), $\mathbf{W}$ (weak until), $\mathbf{F}$ (eventually) and $\mathbf{G}$ (always) (Magee and Kramer, 2006). A property P is specified using an FLTL formula Φ as follows:

$$\textbf{assert } P = \Phi.$$

Property specification was recognized as a difficult task requiring expert knowledge in formal methods, especially in temporal logic. However, according to the rigorous analysis performed by Dwyer *et al*. (1999), most of the specifications fall into the category of specification pat-

terns. Therefore, in our work we have looked into the application of property specification patterns to the verification of *Matchmaker* and *Front-agent* middle-agents. In particular, we show how *response properties* that check if a given situation must always be followed by another situation can be applied to the verification of our system.

For the *Matchmaker* middle-agent, we can define a fluent that holds while *Matchmaker* is processing a request from a given *Requester*,

$$\textbf{fluent}\ MM\_PROC\_REQ(r \in \mathcal{R}) = \langle \{request(r)\}, \{tell(r, P \subseteq \mathcal{P})\}\rangle$$

Similarly, we can define a fluent that holds while the system (*Matchmaker* and *Provider*) is processing a request from a given *Requester*. Note that this property captures also the intuition that the requester can finalize the processing immediately (without getting $receive\_reply$ or $refuse\_request$) when there is no registered provider,

$$\textbf{fluent}\ PROC\_REQ(r \in \mathcal{R}) = \langle \{request(r)\}, \{receive\_reply(r, p \in \mathcal{P}), refuse\_request(r, p \in \mathcal{P}), tell(r, \emptyset)\}\rangle$$

Similarly, we can define a fluent that holds while *Front-agent* is processing a request from a given *Requester*,

$$\textbf{fluent}\ FA\_PROC\_REQ(r \in \mathcal{R}) = \langle \{request(r)\}, \{success(r), fail(r)\}\rangle$$

We can also describe the *REGISTERED* fluent that holds while *Provider* is registered with either *Matchmaker* or *Front-agent* (note that the *REGISTERED* fluent holds also for the *Broker* middle-agent),

$$\textbf{fluent}\ REGISTERED(p \in \mathcal{P}) = \langle \{offer(r)\}, \{withdraw(r)\}\rangle$$

Using the $REGISTERED(p \in \mathcal{P})$ properties, we can define an indexed set of properties $MATCHES(r \in \mathcal{R})$ that are true when there is a registered $Provider$ that matches the given $Requester\ r \in \mathcal{R}$,

$$\textbf{assert}\ MATCHES(r \in \mathcal{R}) = \vee_{p \in \mathcal{M}(r)} REGISTERED(p)$$

Using the defined fluents and formulas, we can express two characterizing response properties of a system with *Front-agent*: (i) "If a $Requester$ issues a request to *Front-agent*, then *Front-agent* will eventually reply either with success or failure" and (ii) "If *Requester* issues a request to *Front-agent* and there is at least one matching *Provider* registered with *Front-agent*, then *Front-agent* will eventually reply with success". These properties can be formally defined using FLTL as follows:

$$\textbf{assert}\ FA\_RESPONSE(r \in \mathcal{R}) = \mathbf{G}\,(request(r) \rightarrow (FA\_PROC\_REQ(r)\,\mathbf{U}\,(success(r) \vee fail(r))))$$

$$\textbf{assert}\ FA\_MATCHING\_RESPONSE(r \in \mathcal{R}) = \mathbf{G}\,((request(r) \wedge MATCHES(r)) \rightarrow (FA\_PROC\_REQ(r)\,\mathbf{U}\,success(r))).$$

Different response properties can be defined for the *Matchmaker* middle-agent: "If *Requester* issues a request to *Matchmaker*, then *Matchmaker* will eventually reply with the set of matching providers",

$$\textbf{assert}\ MM\_RESPONSE(r \in \mathcal{R}) = \mathbf{G}\,(request(r) \rightarrow (MM\_PROC\_REQ(r)\,\mathbf{U}\ \vee_{P \subseteq \mathcal{M}(r)} tell(r, P)))$$

We would be tempted to define a response property of the system with *Matchmaker* similar to *Front-agent*: "If there is a matching *Provider* offer already registered with *Matchmaker* when a request is issued by *Requester*, then the request will be served by a matching *Provider*",

$$\textbf{assert}\ MM\_MATCHING\_RESPONSE\_BAD\ (r \in \mathcal{R}) = \mathbf{G}\,((request(r) \wedge MATCHES(r)) \rightarrow (PROC\_REQ(r)\,\mathbf{U}\,receive\_reply(r, p \in \mathcal{P})))$$

Note, however, that this property does NOT hold! Between the time the matching offers are provided to *Requester* by *Matchmaker* and the time when *Requester* decides to contact a matching *Provider*, it may happen that the matching *Provider* decides to cancel the offer by unregistering with *Matchmaker*.

The correct reformulation of this property would require to guarantee that the system will either perform the request and reply accordingly (the action $receive\_reply$) or will indicate explicitly that fulfilling the request was refused (the action $refuse\_request$),

$$\textbf{assert}\ MM\_MATCHING\_RESPONSE(r \in \mathcal{R}) = \mathbf{G}\,((request(r) \wedge MATCHES(r)) \rightarrow (PROC\_REQ(r)\,\mathbf{U}\,\{receive\_reply(r, p \in \mathcal{P}), refuse\_request(r, p \in \mathcal{P})\}))$$

7. Experiments and discussions

We conducted a series of experiments to check the correctness of our models introduced in Section 5. As a side effect, we also recorded the size of the state model expressed as the number of states and transitions, depending on the number of requesters and providers.

7.1. Experimental setup. Firstly we had to express the general models shown in Figs. 1–3 using the FSP/FLTL language supported by the LTSA tool (version 3.0) (Magee and Kramer, 2006). The main difficulty is the encoding of processes indexed with sets in the language supported by LTSA.

Assuming that $\mathcal{S}$ is a set with n elements, the mapping of processes indexed with sets and/or set elements to the FSP notation supported by LTSA follows the guidelines: (i) an index $s \in \mathcal{S}$ is encoded as $[s]$; (ii) an index $S \subseteq \mathcal{S}$ is encoded as $[s_1] \ldots [s_n]$ such that $s_i = 1$ if $i \in S$ and $s_i = 0$ if $i \notin S$. For example, $ResolveReq(2, \{1\}, \{1, 3\})$ is mapped to `ResolveReq[2][1][0][1][0][1]` and $tell(1, \{2, 3\})$ is mapped to `tell[1][0][1][1]`.

Note that a mapping can be defined such that the size of the resulting FSP specification is linear in the product of number of providers with the number of requesters[5]. This is an important desiderata to make the resulting FSP specification of a practical value.

Proposition 2. *Let $m = |\mathcal{R}|$ and $n = |\mathcal{P}|$. The systems with* Matchmaker*,* Front-agent *and* Broker *shown in Figs. 1–3 can be mapped to the FSP language supported by the LTSA tool such that the size of the resulting specification is $O(m \times n)$.*

Proof. The result is a consequence of the following two observations.

First note that the mapping of set indexed names of processes and actions produces new names of size linear with m and n.

Second, the application of the following mapping rules produces parts of FSP specification that clearly have a size of $O(m \times n)$.

If $Proc(S \subseteq \mathcal{S})$ is a set indexed process and $|\mathcal{S}| = n$, then the construct

$$Proc(S \subseteq \mathcal{S}) = (\textbf{while}\ S \neq \emptyset\ action(s \in S) \ldots)$$

is mapped to (here assuming $n = 3$)

```
Proc[s1:0..1][s2:0..1][s3:0..1] = (
  while s1 == 1 action[1] ... |
  while s2 == 1 action[2] ... |
  while s3 == 1 action[3] ...).
```

Note that the size of the resulting specification is $O(n)$.

If $Proc_i(S \subseteq \mathcal{S})$, $i = 1, 2$, are set indexed processes and $|\mathcal{S}| = n$, then the construct

$$Proc_1(S \subseteq \mathcal{S}) = (\textbf{while}\ S \neq \emptyset\ Proc_2(S) \ldots)$$

is mapped to (here assuming $n = 3$)

```
{Proc1[s1:0..1][s2:0..1][s3:0..1] = (
  if s1 == 1 || s2 == 1 || s3 == 1
  then Proc2[s1:0..1][s2:0..1]
  [s3:0..1] ...)
```

Note that the size of the resulting specification is $O(n)$.

If $\mathcal{S}_1$ and $\mathcal{S}_2$ are sets, $|\mathcal{S}_1| = m$ and $|\mathcal{S}_2| = n$, $\mathcal{M} \subseteq \mathcal{S}_1 \times \mathcal{S}_2$ is a relation, then the construct

$$Proc_1(s \in \mathcal{S}_1, S \subseteq \mathcal{S}_2) = (\textbf{if}\ \ \mathcal{M}(s) \cap S = \emptyset\ \textbf{then}\ \ldots\ \textbf{else}\ \ Proc_2(\mathcal{M}(s) \cap S) \ldots)$$

is mapped to (here assuming $m = 2$, $n = 3$ and $\mathcal{M} = \{(1, 2), (1, 3), (2, 1), (2, 2)\}$)

```
Proc1[s:1..2][s1:0..1][s2:0..1]
[s3:0..1] = (
  if s == 1 then
     if s2 == 0 && s3 == 0 then ...
     else Proc2[0][s2][s3] ...
  else if s == 2 then ...
     if s1 == 0 && s2 == 0 then ...
     else Proc2[s1][s2][0] ...).
```

Note that the size of the resulting specification is $O(m \times n)$. ■

7.2. Results and discussion. In the experiments we considered three systems composed of: (i) n requesters and $n+1$ providers, $2 \leq n \leq 5$; (ii) ones middle-agent per system (*Matchmaker*, *Front-agent* and respectively, *Broker*); and, (iii) the matching relation $\mathcal{M} = \{(i, i+1) | 1 \leq i \leq n\} \cup \{(i, i+2) | 1 \leq i \leq n-1\} \cup \{(n, 1)\}$.

We utilized model checking tools provided by LTSA to analyze the resulting FSP models of these systems that contain *Matchmaker*, *Front-agent* and *Broker* middle-agents. The result of this analysis shows that these models are free of deadlocks, which is consistent with theoretical results (Section 5.4, Proposition 1). Sizes in terms of the number of states and transitions of their corresponding labeled transition systems are presented in Tables 2–4.

For example, in Fig. 4 we present the encoding of the *Front-agent* system from Fig. 2 using the FSP notation supported by the LTSA tool. Note that this system contains one *Front-agent*, two *Requester* agents and three *Provider* agents. The matching function is defined as follows: $\mathcal{M} = \{(1, 2), (1, 3), (2, 3), (2, 1)\}$, i.e., *Requester1* will match with *Provider2* and *Provider3*, and *Requester2* will match with *Provider3* and *Provider1*. Referring to Fig. 4, the fact that the request of *Requester1* must match with *Provider2* and *Provider3* is modeled as follows:

```
ResolveReq[r:1..2][p1:0..1][p2:0..1]
[p3:0..1] =
  if r == 1 then
    if p2 == 0 && p3 == 0 then
      (fail[r]
```

[5] Do not confuse the size of the FSP specification (i.e., the size of the FSP code measured, for example, as the number of nodes of its syntax tree) with the size of the LTS corresponding to this specification.

Table 2. LTS size of the system with the *Matchmaker* middle-agent.

# requesters	# providers	# states	# transitions	# states after minimization
2	3	608	2272	376
3	4	3392	14720	2064
4	5	166400	1010176	N/A
5	6	> 1000000	> 7000000	N/A

Table 3. LTS size of the system with the *Front-agent* middle-agent.

# requesters	# providers	# states	# transitions	# states after minimization
2	3	56	92	52
3	4	160	268	148
4	5	416	704	384
5	6	1024	1744	944

```
    -> Frontagent[p1][p2][p3])
else ContactProvider[r][0][p2][p3]
...
```

This means that when $r = 1$, i.e., *Front-agent* received a request from *Requester1* and neither *Provider2* nor *Provider3* are registered (i.e., $p2 = 0$ and $p3 = 0$), *Front-agent* will return failure, otherwise it will proceed to serving the request.

7.3. Experiments with properties verification. We also conducted model checking experiments with LTSA to check our models of *Matchmaker* and *Front-agent* against the FLTL qualitative properties introduced in Section 6. We updated our mapping framework with the mapping of FLTL formulas. For example, the response properties $MM\_RESPONSE$, $MM\_MATCHING\_RESPONSE\_BAD$, as well as $MM\_MATCHING\_RESPONSE$ for *Matchmaker* are shown in Fig. 5 (assuming there are four requesters, five providers and $\mathcal{M}(2) = \{3, 4\}$)[6].

We utilized the LTSA tool to check the property $MM\_MATCHING\_RESPONSE\_BAD$ for $r = 2$. We obtained the trace presented in Fig. 6 that shows that the property does not hold.

More precisely, taking into account that in this system there are four requesters and five providers, this trace points out to the following system execution scenario: Providers 1, 2 and 3 register their offers with *Matchmaker*; Requester 2 queries *Matchmaker*, which consequently responds with a single matching offer – $\{3\}$ (remember that in this example Requester 2 matches with Providers 3 and 4); Provider 3 then cancels the offer; finally, Requester 2 contacts Provider 3, which however, has to refuse the request, as it was canceled before. Summarizing, when `request[2]` occurred, `MATCHES[2]` was true (as `REGISTERED[3]` was true), while the fluent `PROC_REQ[2]` ceased to be true before the occurrence of an event `receive_reply[2][p]` with $1 \leq p \leq 5$, as it was required by the property `MM_RESPONSE2_BAD`.

Our conclusion after applying LTSA to checking FSP models of middle-agents is that, despite the usefulness of model checking to verify simple models of middle agents, the technique is limited for at least two reasons: (i) the approach is not general because it is only able to check specific instances of systems with a given number of agents and a specific matching relation rather than general models; (ii) the state-space explosion problem hinders the analysis of systems with a larger number of states.

Nevertheless, we think that the developed models are useful to better understand available types of middle-agents and to enable a theoretical study of their qualitative properties, which would be obviously more robust than the experimental investigation using model checking.

8. Conclusions

Our work sheds light on the modeling of software systems incorporating middle-agents using process algebra for defining the semantics of the different actors in such applications. In particular, we proposed a formal framework based on FSP process algebra and FLTL temporal logic for the modeling and verification of systems that contain requesters, providers and middle-agents. We applied this framework to model and verify systems with three types of middle-agents: *Matchmaker*, *Front-agent*, and *Broker*. For each system, we defined a sample set of qualitative properties expressed in FLTL and we checked the resulting models against these properties with the help of the LTSA analysis tool. While this work might look incomplete with respect (i) to the coverage of the set of nine types of middle-agents that were identified in the literature, and (ii) to considering only those properties that are formally expressible using FLTL, it still has a sen-

[6]FSP and FLTL models used in experiments are available at `http://software.ucv.ro/~cbadica/fsp/amcs10_models.zip`.

Table 4. LTS size of the system with the *Broker* middle-agent.

# requesters	# providers	# states	# transitions	# states after minimization
2	3	293	464	281
3	4	1788	2997	1718

```
// M ={(1,2),(1,3),(2,1),(2,2)}
Frontagent = Frontagent[0][0][0],
Frontagent[p1:0..1][p2:0..1][p3:0..1] = (
  request[r:1..2] -> ResolveReq[r][p1][p2][p3] |
  when p1 == 0 offer[1] -> Frontagent[1][p2][p3] |
  when p2 == 0 offer[2] -> Frontagent[p1][1][p3] |
  when p3 == 0 offer[3] -> Frontagent[p1][p2][1]
),
ResolveReq[r:1..2][p1:0..1][p2:0..1][p3:0..1] =
  if r == 1 then
    if p2 == 0 && p3 == 0 then (fail[r] -> Frontagent[p1][p2][p3])
    else ContactProvider[r][0][p2][p3]
  else if r == 2 then
    if p1 == 0 && p2 == 0 then (fail[r] -> Frontagent[p1][p2][p3])
    else ContactProvider[r][p1][p2][0],
ContactProvider[r:1..2][p1:0..1][p2:0..1][p3:0..1] = (
    when p1 == 1 fragent_req[1] -> fragent_reply[1] -> success[r] -> Frontagent[p1][p2][p3] |
    when p2 == 1 fragent_req[2] -> fragent_reply[2] -> success[r] -> Frontagent[p1][p2][p3] |
    when p3 == 1 fragent_req[3] -> fragent_reply[3] -> success[r] -> Frontagent[p1][p2][p3]).

Requester = (
  send_request -> WaitReply),
WaitReply = (
  {success, fail}-> Requester).

Provider =
  (offer -> ProcessRequests),
ProcessRequests = (
   receive_request -> send_reply -> ProcessRequests).

||Reequester1 = Requester/{request[1]/send_request,fail[1]/fail,success[1]/success}.
||Reequester2 = Requester/{request[2]/send_request,fail[2]/fail,success[2]/success}.

||Provider1 = Provider/{offer[1]/offer,fragent_req[1]/receive_request,fragent_reply[1]/send_reply}.
||Provider2 = Provider/{offer[2]/offer,fragent_req[2]/receive_request,fragent_reply[2]/send_reply}.
||Provider3 = Provider/{offer[3]/offer,fragent_req[3]/receive_request,fragent_reply[3]/send_reply}.

||System = (Frontagent || Reequester1 || Reequester2 || Provider1 || Provider2 || Provider3).
```

Fig. 4. Encoding of the *Front-agent* model with $n = 2$ for the LTSA tool.

```
const False = 0
const R = 4
const P = 5
fluent REGISTERED[p:1..P] =
  <{offer[p]},{withdraw[p]}> initially False
fluent PROC_REQ[r:1..R] =
  <{request[r]},{receive_reply[r][p:1..P],refuse_request[r][p:1..P]}> initially False
fluent MATCHMAKER_PROC_REQ[r:1..R] =
  <{request[r]},{tell[r][p1:0..1][p2:0..1][p3:0..1][p4:0..1][p5:0..1]}> initially False
assert MATCHES2 =
  (REGISTERED[3] || REGISTERED[4])
assert MM_RESPONSE2 =
  [](request[2] -> (MATCHMAKER_PROC_REQ[2] U {tell[2][p1:0..1][p2:0..1][p3:0..1][p4:0..1][p5:0..1]}))
assert MM_MATCHING_RESPONSE2_BAD =
  [](request[2] && MATCHES2 -> (PROC_REQ[2] U receive_reply[2][p:1..P]))
assert MM_MATCHING_RESPONSE2 =
  [](request[2] && MATCHES2 -> (PROC_REQ[2] U {receive_reply[2][p:1..P],refuse_request[2][p:1..P]}))
```

Fig. 5. Encodings of system properties for the LTSA tool.

```
Violation of LTL property: @MM_RESPONSE2_BAD
Trace to terminal set of states:
    offer.1
    offer.2
    offer.3     REGISTERED.3
    request.2       REGISTERED.3 && PROC_REQ.2
    tell.2.0.0.1.0.0        REGISTERED.3 && PROC_REQ.2
    withdraw.3      PROC_REQ.2
    send_req_to_provider.2.3        PROC_REQ.2
    refuse_request.2.3
    offer.3     REGISTERED.3
    request.2       REGISTERED.3 && PROC_REQ.2
    tell.2.0.0.1.0.0        REGISTERED.3 && PROC_REQ.2
    offer.4     REGISTERED.3 && REGISTERED.4 && PROC_REQ.2
    request.3       REGISTERED.3 && REGISTERED.4 && PROC_REQ.2
    tell.3.0.0.0.1.0        REGISTERED.3 && REGISTERED.4 && PROC_REQ.2
    offer.5     REGISTERED.3 && REGISTERED.4 && PROC_REQ.2
    request.4       REGISTERED.3 && REGISTERED.4 && PROC_REQ.2
    tell.4.1.0.0.0.1        REGISTERED.3 && REGISTERED.4 && PROC_REQ.2
    withdraw.3      REGISTERED.4 && PROC_REQ.2
    request.1       REGISTERED.4 && PROC_REQ.2
Cycle in terminal set:
    tell.1.0.1.0.0.0        REGISTERED.4 && PROC_REQ.2
    send_req_to_provider.1.2        REGISTERED.4 && PROC_REQ.2
    receive_reply.1.2       REGISTERED.4 && PROC_REQ.2
    request.1       REGISTERED.4 && PROC_REQ.2
LTL Property Check in: 14593ms
```

Fig. 6. Trace of property verification using LTSA.

se of completeness by a full treatment of the modeling-specification-verification cycle of a software system. In the future, we intend to (i) model more complex systems containing all types of middle-agents using a compositional approach; (ii) extend the framework for properties specification and verification with a comprehensive analysis of properties specific to all nine types of middle-agents.

Acknowledgment

The work reported in this paper was partly supported by (i) the research project SCIPA: *Servicii software semantice de Colaborare si Interoperabilitate pentru realizarea Proceselor Adaptive de business* between the Software Engineering Department, University of Craiova, Romania, and the National Authority for Scientific Research, Romania, and partly by (ii) the research project *Agent-Based Service Negotiation in Computational Grids* between the Systems Research Institute, Polish Academy of Sciences, and the Software Engineering Department, University of Craiova, Romania.

We would like to thank the guest editors, Prof. Antoni Ligęza and Dr. Grzegorz J. Nalepa, for the effort made to prepare this special section and also the anonymous reviewers for their very useful feedback to improve the preliminary version of this paper.

References

Alagar, V. and Holliday, J. (2002). Agent types and their formal descriptions, *Technical Report COEN-2002-09-19A*, Santa Clara University, Santa Clara, CA.

Bergstra, J., Ponse, A. and Smolka, S. (2001). *Handbook of Process Algebra*, Elsevier Science, Amsterdam.

Brazier, F.M.T., Cornelissen, F., Gustavsson, R., Jonker, C.M., Lindeberg, O., Polak, B. and Treur, J. (2004). Compositional verification of a multi-agent system for one-to-many negotiation, *Applied Intelligence* **20**(2): 95–117.

Bădică, A. and Bădică, C. (2008a). Conceptualizing interactions with matchmakers and front-agents using formal verification methods, *in* D. Dochev, M. Pistore and P. Traverso (Eds.), *Proceedings of the 13th International Conference on Artificial Intelligence: Methodology, Systems, Applications (AIMSA 2008)*, Lecture Notes in Artificial Intelligence, Vol. 5253, Springer-Verlag, Berlin, pp. 390—394.

Bădică, A. and Bădică, C. (2008b). Formal specification of matchmakers, front-agents, and brokers in agent environments using FSP, *Proceedings of the 6th International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems (MSVVEIS 2008), Barcelona, Spain,* pp. 9–18.

Bădică, A. and Bădică, C. (2008c). Formalizing agent-based english auctions using finite state process algebra, *Journal of Universal Computer Science* **14**(7): 1118–1135.

Bădică, A. and Bădică, C. (2008d). Modeling interactions in agent-based English auctions with matchmaking capabilities, *in* C. Bădică, G. Mangioni, V. Carchiolo and D.D. Burdescu (Eds.), *Intelligent Distributed Computing, Systems and Applications (IDC 2008)*, Studies in Computational Intelligence, Vol. 162, Springer-Verlag, Berlin, pp. 45–54.

Bădică, A. and Bădică, C. (2008e). Specification and verification of agent interactions in matchmaking processes using FSP

and FLTL, *Proceedings of the 23rd International Symposium on Computer and Information Sciences (ISCIS 2008), Istanbul, Turkey*, pp. 1–6.

Bădică, A. and Bădică, C. (2009a). Formalizing agent interactions in matchmaking processes using FSP and FLTL, *Scientific Bulletin of "Politehnica" University of Timisoara, Transactions on Automatic Control and Computer Science* **54(68)**(1): 13–18.

Bădică, A. and Bădică, C. (2009b). Specification and verification of an agent-based auction service, *in* G.A. Papadopoulos, W. Wojtkowski, G. Wojtkowski, S. Wrycza, and J. Zupancic (Eds.) *Information System Development. Towards a Service Provision Society (ISD 2008)*, Springer-Verlag, New York, NY, pp. 239–248.

Bădică, A., Bădică, C. and Liţoiu, L. (2003). Role activity diagrams as finite state processes, *Proceedings of the 2nd International Symposium on Parallel and Distributed Computing (ISPDC 2003), Ljubljana, Slovenia*, pp. 15–22.

Bădică, A., Bădică, C. and Liţoiu, L. (2007). Middle-agents interactions as finite state processes: Overview and example, *Proceedings of the 16th IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2007), Paris, France*, pp. 12–17.

Bădică, A., Bădică, C., Popescu, E. and Scafes, M. (2009). A process algebraic framework for service coordination, *Proceedings of the 5th International Symposium on Applied Computational Intelligence and Informatics (SACI 2009), Timişoara, Romania*, pp. 515–520.

Bădică, C., Ganzha, M. and Paprzycki, M. (2007). Developing a model agent-based e-commerce system, *in* J. Lu, G. Zhang and D. Ruan (Eds.), *E-Service Intelligence: Methodologies, Technologies and Applications*, Studies in Computational Intelligence, Vol. 55, Springer-Verlag, Berlin, pp. 555–578.

Clarke, E.M., Emerson, E.A. and Sistla, A.P. (1986). Automatic verification of finite-state concurrent systems using temporal logic specifications, *ACM Transactions on Programming Languages and Systems* **8**(2): 244–263.

Decker, K., Sycara, K.P. and Williamson, M. (1997). Middle-agents for the internet, *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI'97), Nagoya, Aichi, Japan*, Vol. 1, pp. 578–583.

D'Ippolito, N., Fischbein, D., Chechik, M. and Uchitel, S. (2008). MTSA: The modal transition system analyser, *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering, ASE'08, L'Aquila, Italy*, pp. 475–476.

Dobriceanu, A., Bîscu, L., Bădică, A. and Bădică, C. (2009). The design and implementation of an agent-based auction service, *International Journal of Agent Oriented Software Engineering* **3**(2/3): 188–211.

Dwyer, M.B., Avrunin, G.S. and Corbett, J.C. (1999). Patterns in property specifications for finite-state verification, *Proceedings of the 21st International Conference on Software Engineering (ICSE 1999), Los Angeles, CA, USA*, pp. 411–420.

Esterline, A., Rorie, T. and Homaifar, A. (2006). A process-algebraic agent abstraction, *in* C.A. Rouff, M. Hinchey, J. Rash, W. Truszkowski and D. Gordon-Spears (Eds.), *Agent Technology from a Formal Perspective*, NASA Monographs in Systems and Software Engineering, Springer-Verlag, London, pp. 88–137.

Fasli, M. (2007). *Agent Technology for E-Commerce*, John Wiley & Sons, Chichester.

Fokkink, W. (2007). *Texts in Theoretical Computer Science*, EATCS Series, Springer-Verlag, Berlin/Heidelberg.

Foster, H., Uchitel, S., Magee, J. and Kramer, J. (2006). LTSA-WS: A tool for model-based verification of web service compositions and choreography, *in* L.J. Osterweil, H.D. Rombach and M.L. Soffa (Eds.), *Proceedings of the 28th International Conference on Software Engineering (ICSE 2006), Shanghai, China*, pp. 771–774.

Goh, S., Chhetri, M.B. and Kowalczyk, R. (2007). JADE-FSM-engine: A deployment tool for flexible agent behaviours in JADE, *Proceedings of the 2007 IEEE/WIC/ACM International Conference on Intelligent Agent Technology, IAT'2007, Silicon Valley, CA, USA*, pp. 524–527.

Hennicker, R. and Ludwig, M. (2005). Property-driven development of a coordination model for distributed simulations, *in* M. Steffen and G. Zavattaro (Eds.), *Formal Methods for Open Object-Based Distributed Systems*, Lecture Notes in Computer Science, Vol. 3535, Springer, Berlin/Heidelberg, pp. 290–305.

Hoare, C.A.R. (1985). *Communicating Sequential Processes*, Prentice Hall International Series in Computer Science, Upper Saddle River, NJ.

Hristozova, M. and Sterling, L. (2003). Experiences with ontology development for value-added publishing, *in* S. Cranefield, T.W. Finin, V.A.M. Tamma and S. Willmott (Eds.), *Proceedings of the International Workshop on Ontologies in Agent Systems (OAS 2003)*, CEUR Workshop Proceedings, Vol. 73, pp. 17–24.

Klusch, M. and Sycara, K.P. (2001). Brokering and matchmaking for coordination of agent societies: A survey, *in* A. Omicini, F. Zambonelli, M. Klusch and R. Tolksdorf (Eds.), *Coordination of Internet Agents: Models, Technologies, and Applications*, Springer, Berlin/Heidelberg/New York, NY, pp. 197–224.

Magee, J. and Kramer, J. (2006). *Concurrency. State Models and Java Programs*, 2nd Edn., John Wiley & Sons, Chichester.

Mbala, A., Padgham, L. and Winikoff, M. (2006). Design options for subscription managers, *in* G.A. Vouros and T. Panayiotopoulos (Eds.), *Agent-Oriented Information Systems III*, Lecture Notes in Computer Science, Vol. 3529, Springer, London, pp. 259–274.

Merayo, M., Núñez, M., and Rodríguez, I. (2007). Specification of multi-agent systems by using EUSMs, *Proceedings of the International Symposium on Fundamentals of Software Engineering (FSEN'2007)*, Lecture Notes in Computer Science, Vol. 4767, Springer, Berlin/Heidelberg/New York, NY, pp. 318–333.

Miller, T. and McBurney, P. (2007). Using constraints and process algebra for specification of first-class agent interaction protocols, *Engineering Societies in the Agents World VII (ESAW 2006)*, Lecture Notes in Computer Science, Vol. 4457, Springer, Berlin/Heidelberg/New York, NY, pp. 245–254.

Milner, R. (1999). *Communicating and Mobile Systems: The π-Calculus*, Cambridge University Press, Cambridge.

Podorozhny, R.M., Khurshid, S., Perry, D.E. and Zhang, X. (2007). Verification of multi-agent negotiations using the alloy analyzer, *Proceedings of the 6th International Conference Integrated Formal Methods (IFM'2007)*, Lecture Notes in Computer Science, Vol. 4591, Springer, Berlin/Heidelberg/New York, NY, pp. 501–517.

Rahimi, S., Cobb, M., Ali, D. and Petry, F. (2002). A modeling tool for intelligent-agent based systems: Api-calculus, *in* V. Loia (Ed.), *Soft Computing Agents: A New Perspective for Dynamic Systems*, Frontiers in Artificial Intelligence and Applications, IOS Press, Amsterdam, pp. 165–186.

Rouff, C., Rash, J., Hinchey, M. and Truszkowski, W. (2006). Formal methods at NASA Goddard Space Flight Center, *in* C.A. Rouff, M. Hinchey, J. Rash, W. Truszkowski and D. Gordon-Spears (Eds.), *Agent Technology from a Formal Perspective*, NASA Monographs in Systems and Software Engineering, Springer-Verlag, London, pp. 287–309.

Wang, F. (2002). Self-organising communities formed by middle agents, *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS'02, Bologna, Italy*, pp. 1333–1339.

Wong, H.C. and Sycara, K. (2000). A taxonomy of middle-agents for the Internet, *Proceedings of the 4th International Conference on MultiAgent Systems (ICMAS-2000), Boston, MA, USA*, pp. 465–466.

Xu, D.-X., El-Ariss, O., Xu, W.-F. and Wang, L.-Z. (2009). Aspect-oriented modeling and verification with finite state machines, *Journal of Computer Science and Technology* **24**(5): 949–961.

Yarom, I., Rosenschein, J.S. and Goldman, C.V. (2003). The role of middle-agents in electronic commerce, *IEEE Intelligent Systems* **18**(6): 15–21.

Zhang, P., Muccini, H. and Li, B. (2010). A classification and comparison of model checking software architecture techniques, *Journal of Systems and Software* **83**(5): 723–744.

Amelia Bădică holds a Ph.D. in economics and she currently works as a senior lecturer at the Business Information Systems Department, University of Craiova, Romania. She has specialization in management information systems obtained at Binghamton University, USA. Her research interests cover the application of expert systems, softtware engineering and Web technologies in business and management. She has co-authored several papers in journals and conference proceedings on these subjects. She has been involved as the principal investigator in a research project concerning data extraction from the Web.

Costin Bădică holds a Ph.D. in computer science, and in 2006 he received the title of a professor of computer science from the University of Craiova. He is currently with the Department of Software Engineering, Faculty of Automatics, Computers and Electronics of the University of Craiova, Romania. In 2001 and 2002 he was a post-doctoral fellow with the Department of Computer Science, King's College London, UK. His research interests are at the intersection of artificial intelligence, distributed systems and software engineering. He has authored and co-authored more than 100 publications related to these topics in the form of journal articles, book chapters and conference papers. He has prepared special journal issues and co-edited four books in Springer's *Studies in Computational Intelligence* series. He co-initiated and is co-organizing the Intelligent Distributed Computing (*IDC*) series of international conferences held annually. He is a member of the editorial boards of four international journals. He has also served as programme committee member of numerous international conferences.

Received: 12 April 2010
Revised: 21 November 2010