

# A Requirement Specification Language for Configuration Dynamics of Multi-Agent System

Mehdi Dastani, Catholijn Jonker, and Jan Treur

Vrije Universiteit Amsterdam, Department of Artificial Intelligence  
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands  
{jonker, treur}@cs.vu.nl  
URL: <http://www.cs.vu.nl/~jonker,~treur>

**Abstract.** In agent-mediated applications, the configuration of the multi-agent system often changes due to the creation and the deletion of agents. The behaviour of such systems on the one hand depends on the structural dynamics of the system configuration, but on the other hand consists of the informational dynamics of the configuration. To specify and verify the properties of the system, including its configuration dynamics, a requirement language is needed that is capable to express those properties. In this paper, we discuss the configuration dynamics properties of multi-agent systems and define a language by means of which those properties can be specified. A prototypical scenario for an agent-mediated system is discussed and some important requirements for this system are specified.

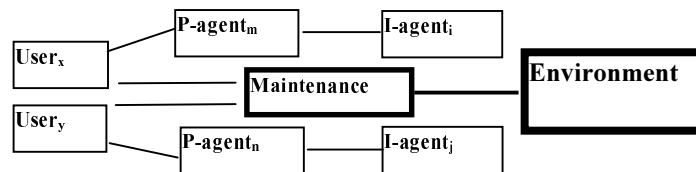
## 1 Introduction

Requirements Engineering is today a well-studied field of research within Software Engineering; e.g., [2], [14], [11]. Requirements describe the required properties of a system (this may include the functions of the system, structure of the system, static and dynamic properties). In recent years requirements engineering for distributed and agent systems has been studied in some depth, e.g., in [3], [5], [9]. In applications to agent-based systems, the dynamics or behaviour of the system plays an important role in description of the successful operation of the system. To be able to express requirements unambiguously, and to support verification by automated tools, formal requirements specification languages are important.

Dynamics of agent-based systems can take different forms. Depending on the type of dynamics, different demands are imposed on the expressivity of a requirements specification language. A simple form of requirements for dynamics (close to functional requirements) expresses reactive types of behaviour. However, combinations of pro-active and reactive behaviour can require more complicated requirements expressions. Most types of behaviour of this type can be expressed using rather standard forms of temporal logic; e.g., [13], [7], [8]. Further complications arise if the evolution of a system over time is taken into account; then temporal logics of the more standard types do not suffice. Examples of behaviour of this type are relative adaptive

behaviour (e.g., ‘exercise improves skill’), in which two different possible histories have to be compared, and self-modifying behaviour: for example, the behaviour of one of the agents (e.g., a creation action) leads to a different system structure (e.g., with an additional agent). This paper addresses specifications of requirements for the latter type of dynamics.

In order to illustrate this type of dynamics, consider multi-agent mediating systems such as brokering systems, match making systems, and search engines, which are important electronic commerce applications. In such applications, a user interacts with the system, receives its personal assistant, which in turn interacts with available intermediate and task specific agents to perform the user task. Note that in these systems agents (personal assistant and task specific agents) are created and removed from the system. In this paper, we propose an abstract multi-agent architecture for such mediating systems. According to this architecture, which is illustrated in Figure 1, mediating systems consist of a central maintenance agent, which is responsible for system configuration, an environment component, which represents the system environment including all components of the system except itself and is responsible for the realization of system modifications, and zero or more agents that can be distinguished in two types: personal assistant agents (P-agents), which assists their users to achieve their tasks, and intermediate task specific agents (I-agents such as negotiator, matchmakers, and auctioneer), which are responsible for the actual performance of the user tasks.



**Fig. 1.** The generic architecture of the agent mediating systems

Initially, the maintenance agent interacts with users (through dashed links) and determines the required modification of system configuration. The maintenance agent communicates then with the environment (through the bold link) and order to modify the system configuration. The environment implements the given order through which the system configuration is modified. The users can then delegate their tasks to their configured personal assistants, which in turn achieve the tasks by interacting with the configured task specific intermediate agents (through the remaining links). For example, a user who is willing to buy a car communicates with the maintenance agent and requires a car brokering system. The maintenance agent specifies a car brokering system consisting of a car broker agent (which may already exist) and a personal assistant agent for the user (other personal assistant agents may exist already since the buyer has been registered) in such a way that the car broker agent and personal assistant agents can interact. The specified system configuration is then communicated to the environment component, which realizes the required configuration. The user can now delegates its car-buying task to its assigned personal assistant agent, which in turn

can interact with the car broker agent to achieve the delegated task. The user can decide to terminate the broking system in which case the personal assistant agent can be either deleted from the system or set idle. The car broker agent may also be either deleted or set idle if there are no other car related personal assistant agents.

One essential property of these agent-mediating systems is their dynamic configurations: new agents are created and removed from the system. A correctly behaving system depends therefore on its configuration during its execution. For this reason, it is important to specify configuration properties of such systems during their executions. This is the focus of the paper. In particular, we will develop a trace requirement language that can express, beside functional and informational requirements, those requirements that are concerned with the system configuration during its execution traces. Based on this language, we will then identify a number of requirement types concerning the configuration of agent-mediated systems. A scenario for an agent-mediating system will motivate these requirements.

## 2 Requirement Language

In this section, we first define an ordered-sorted language, called Design Language (*DL*), the expressions of which specify the configurations of agent-mediating system. These expressions will be used to specify the required system configuration. Then, based on the *DL* expressions we define another ordered-sorted language, called Design Action Language (*DAL*), the expressions of which modify system configuration, i.e. the expressions indicate actions that change one system configuration to another. For example, the maintenance agent in our example generates such expressions. In order to modify the system configuration, the modification actions need to be communicated. In our example, the maintenance agent communicates such modification actions with the environment component. In order to specify these communication expressions, we define a third ordered-sorted language, called Communication Action Language (*CAL*) that uses the *DAL* expressions and generates such communication expressions. Finally, the expressions of these languages will be used to define the Trace Requirement Language (*TRL*) the expressions of which formalize properties of not only the functional behaviour of the system, but also the configuration of the system during execution traces.

Note that the first three languages are object languages in the sense that they are used by the components in the agent-mediating systems to specify and modify the configuration of the system, while the trace requirement language is a meta-language in the sense that it is used (by an external observer) to specify the (information and configuration) properties of the system behaviour. Consequently, the first three object languages presuppose certain type of multi-agent system architecture since their expressions are about system ingredients. In this paper, we concentrate on systems specified according to some form of compositional architecture).[1]. These systems consist of a number of components that interact with each other through certain types of connections. Components may or may not be composed of other components. Components that are not composed are sometimes called primitive components. These com-

ponents are defined in terms of certain ingredients such as input and output interfaces, control loop, signatures, embedded components (for non-primitive components), and knowledge bases (for primitive components). The to be formalized configuration properties are about these ingredients and their structural relations. Although we will not give an exhaustive list of all ingredients that may be used in these types of systems, we mention and use the most important ingredients. Thus we formalise systems on the basis of some form of component-based or compositional architecture, as for example (but certainly not exclusively) is possible within DESIRE, (DEsign and Specification of Interacting REasoning components; cf. [1]) The reader should note that the proposed object languages could be easily reformulated for different types of systems.

## 2.1 Design Language (*DL*)

The sorts in *DL* identify the ingredients in such systems. Although an exhaustive enumeration of all sorts that are used is out of scope of this paper, some important sorts are mentioned in table 1.

**Table 1.** Examples of sorts in *DL*

Sort	Description
SP	sort for system components; part names identify a specific system component in the hierarchical system structure
SP <sub>prim</sub>	sort for primitive system components
SP <sub>Interface</sub>	sort for component interfaces (i.e. input or output)
SCON	sort for connections between system components
SSIG	sort for signatures used by components
SKB	sort for knowledge bases of primitive system components

These sorts can be ordered, like for example  $SP_{prim} < SP$ . For all sorts *S* a set of constants (i.e.  $\{c \mid c \text{ is a constant of sort } S\}$ ) and a set of variables (i.e.  $\{x:S \mid x \text{ is a variable of sort } S\}$ ) are assumed. Also, a set of *n*-ary functions (i.e.  $\{f^n \mid f^n \text{ is an } n\text{-ary function for any } n\}$ ) is assumed. Given the above ingredients, the terms of the design language can be defined as follows:

1. If *c* is a constant of sort *S*, then *c* is a term of sort *S*.
2. If *x*:*S* is a variable of sort *S*, then *x*:*S* is a term of sort *S*.
3. If  $f:S_1 \times \dots \times S_n \rightarrow S$  is a *n*-ary function and  $t_i$  is a term of sort  $S_i$  ( $i=1, \dots, n$ ), then  $f(t_1, \dots, t_n)$  is a term of sort *S*.

These terms refer to ingredients of the system, which can be related to each other according to some certain relations. These relations are denoted by sorted predicate symbols. Some important predicates are mentioned in table 2.

**Table 2.** Examples of predicates in *DL*

Predicate	Description
exists_comp:SP	component exists
sub:SP×SP	subcomponent relation between system components
connected_to:SP×SP×SCON	components connected by connection
has_input_sign:SP×SSIG	component uses input signature
has_private_sign:SP×SSIG	component uses private signature
has_KnowBase:SP <sub>prim</sub> ×SKB	primitive component has knowledge base

Based on these sorted predicates, the formulae of the design language *DL* can be defined as follows:

1. If  $P:S_1 \times \dots \times S_n$  is a  $n$ -ary predicate and  $t_i$  ( $i=1, \dots, n$ ) is a term of sort  $S_i$ , then  $P(t_1, \dots, t_n)$  is a formula of *DL*.
2. If  $E_1$  and  $E_2$  are formulae of *DL* and  $x$  is a variable of sort  $S$ , then  $E_1 \wedge E_2$ ,  $E_1 \vee E_2$  and  $\neg E_1$  are formulae of *DL*.

Let  $sig_x$  be a signature such as ACL. “ $Connected\_to(agent_{PA}, agent_{IA}, from\_to_3) \wedge has\_Input\_Signature(agent_{PA}, sig_x)$ ” is a design formula saying “the personal assistance agent  $agent_{PA}$  is connected to the maintenance agent  $agent_{IA}$  by connection  $from\_to_3$  and agent  $agent_{PA}$  uses input signature  $sig_x$ ”.

## 2.2 Design Action Language (DAL)

*DAL* imports all terms and formulae from *DL* as *terms* and generates new terms (no formulae) denoting plans to modify a system configuration. Note that importing *DL* formulae as terms implies that these terms refer to the structural configurations of agent-mediated system. The sorts in *DAL* are given in table 3.

**Table 3.** Sorts of *DAL*

Sort	Description
DEAT	imported design atoms from <i>DL</i>
DEFOR	imported formula from <i>DL</i>
DEFOR <sup>+</sup>	imported positive design formulae (design atoms or conjunction of positive design formulae)
ACTION	actions
SIGN	truth values
$S_i$ $i=1, \dots, n$	imported sorts from <i>DL</i>

Note the following subsort relation:  $DEAT < DEFOR^+ < DEFOR$ . Similar to *DL*, for all sorts a set of constants and a set of variables are assumed. In order to import terms and formula from *DL* as terms in *DAL*, we introduce functions corresponding to both the imported functions and predicates. Given  $sort(DL)$ ,  $func(DL)$ , and  $pred(DL)$  be respectively the sets of sorts, functions, and predicates from *DL*, the functions in *DAL* are defined as follows:

1. For all n-ary function  $f \in \text{func}(DL)$  and all sorts  $S, S_1, \dots, S_n \in \text{sort}(DL)$  where  $f: S_1 \times \dots \times S_n \rightarrow S$ ,  $\underline{f}: S_1 \times \dots \times S_n \rightarrow S$  is an imported function in *DAL* (functions  $f$  from *DL* are imported as functions  $\underline{f}$  in *DAL*).
2. For all n-ary predicates  $p \in \text{pred}(DL)$  and all sorts  $S_1, \dots, S_n \in \text{sort}(DL)$  where  $p: S_1 \times \dots \times S_n$ , predicate  $p$  is reformulated as a function  $\underline{p}: S_1 \times \dots \times S_n \rightarrow \text{DEAT}$  and imported in *DAL* (predicates are imported as functions).

Also, logical connectives that operate on *DL* formula are imported as functions.

3.  $\underline{\wedge}: \text{DEFOR}^+ \times \text{DEFOR}^+ \rightarrow \text{DEFOR}^+$
4.  $\underline{\wedge}: \text{DEFOR} \times \text{DEFOR} \rightarrow \text{DEFOR}$
5.  $\underline{\vee}: \text{DEFOR} \times \text{DEFOR} \rightarrow \text{DEFOR}$
6.  $\underline{\neg}: \text{DEFOR} \rightarrow \text{DEFOR}$

Two specific (action related) functions that are used in *DAL* but not imported from *DL* are as follows:

7.  $\text{Add}: \text{DEFOR}^+ \rightarrow \text{ACTION}$
8.  $\text{Delete}: \text{DEAT} \rightarrow \text{ACTION}$

Note that the imported functions in *DAL* are marked (by being underlined) in order to distinguish them from their corresponding functions or predicates from *DL*. Whenever there is no confusion, we will leave out these underlying marks. Given the above functions, the terms of *DAL* can be defined in usual way.

“ $\text{Add}(\text{Connected\_to}(\text{agent}_{PA}, \text{agent}_{IA}, \text{from\_to}_3) \wedge \text{Has\_Input\_Signature}(\text{agent}_{PA}, \text{sig}_d))$ ” is a *DAL* term, which denotes a plan to create a system consisting of  $\text{agent}_{PA}$  and  $\text{agent}_{IA}$  that are connected by connection  $\text{from\_to}_3$  and, moreover,  $\text{agent}_{PA}$  uses the input signature  $\text{sig}_d$ . The function  $\text{add}$  should be interpreted as adding all system elements that occur in the design formula but not yet included in the system, in the given relation.

### 2.3 Design Communication Language (DCL)

*DCL* imports terms from *DAL* and generates expressions by means of which system components can communicate about system configuration and its modification. Any sort from *DAL* is also a sort in *DCL*. For all sorts a set of constants and a set of variables are assumed. Also, any function from *DAL* is imported as a function in *DCL*. Note that there is a one-to-one correspondence between terms from *DCL* and *DAL*. Moreover, a set of predicates is assumed that determines the communication between components about system configuration and its modification. These predicates are like those that are used in *DESIRE*, though we will not enumerate them exhaustively. Some important predicates are given in table 4.

**Table 4.** Sorts of *DAL*

Predicate	Description
to_be_performed:ACTION	design action to be performed
to_be_observed:DEFOR	observe if design formula holds
observation_result:DEFOR×SIGN	design formula has truth value

Based on the defined terms and predicates, the formulae of DCL can be defined as follows:

1. If  $P:S_1 \times \dots \times S_n$  is a  $n$ -ary predicate and  $t_i$  is a term of sort  $S_i$  ( $i=1, \dots, n$ ), then  $P(t_1, \dots, t_n)$  is an atomic formula of the design command language *DCL*.
2. All atomic *DCL* formulae are *DCL* formulae.
3. If  $E$  is an atomic *DCL* formula, then  $\neg E$  is a *DCL* formula.
4. If  $E_1$  and  $E_2$  are *DCL* formulae, then  $E_1 \wedge E_2$  is a *DCL* formula.

“to\_be\_performed(Add(Connected\_to(agent<sub>PA</sub>, agent<sub>IA</sub> from<sub>to3</sub>) ^ has\_Input\_Signature(agent<sub>PA</sub>, sig<sub>2</sub>)))” is a design communication formula, which states that the add-action should be performed.

## 2.4 Trace Requirement Language (*TRL*)

Trace Requirement Language (*TRL*) is defined to specify properties or requirements concerning either the configuration behaviour (e.g. what will be the system configuration if the maintenance agent sends a *DCL* expression to the environment component) or the information behaviour (e.g. what will be the system information state if a component sends an expression to another component) of agent-mediated systems. A trace is a sequence of system states (models) through the time. Thus, *TRL* generates expressions that specify the configuration and information behaviour of a system through time. For this reason, we define terms that denote system states, system traces, information content of system states, and structural configurations of system states. Moreover, we introduce predicates to represent relations between above entries.

Given a domain specific ordered-sorted signature  $\Sigma$  (like those used in *DESIRE*), we denote sorts, functions and predicates from  $\Sigma$  by  $sort(\Sigma)$ ,  $func(\Sigma)$  and  $pred(\Sigma)$ , respectively. Also, we denote sorts, functions and predicates from *DCL* by  $sort(DCL)$ ,  $func(DCL)$  and  $pred(DCL)$ , respectively. The sorts in *TRL* are mentioned in table 5.

**Table 5.** Sorts of *TRL*

Sort	Description
IFOR	information formulae from $\Sigma$
DCFOR	formulae from <i>DCL</i>
M <sub>R</sub>	restricted states or models
M <sub>C</sub>	complete states or models
T	time
Γ	trace
S <sub>i</sub> $i=1, \dots, n$	imported sorts from $\Sigma$ and <i>DCL</i>

We define a new sort  $FOR$  such that  $IFOR < FOR$  and  $DCFOR < FOR$ . Note that  $DCFOR$  formulae express communication about design formulae while  $IFOR$  formulae express communication about information formulae,  $SP_{Comp} < SP$ ,  $SP_{Interface} < SP_{Comp}$ ,  $SP_{Input} < SP_{Interface}$ ,  $SP_{Output} < SP_{Interface}$ . Moreover, for all sorts a set of constants and a set of variables are assumed. For all n-ary function  $f \in func(\Sigma) \cup func(DCL)$  and all sorts  $S, S_1, \dots, S_n \in sort(\Sigma) \cup sort(DCL)$  where  $f: S_1 \times \dots \times S_n \rightarrow S$ ,  $\underline{f}: S_1 \times \dots \times S_n \rightarrow S$  is an imported function in the trace-requirement language (functions  $f$  from  $\Sigma$  or  $DCL$  are imported as functions  $\underline{f}$  in the  $TRL$ ). Also, for all n-ary predicates  $p \in pred(\Sigma)$  and all sorts  $S_1, \dots, S_n \in sort(\Sigma)$  where  $p: S_1 \times \dots \times S_n$ , predicate  $p$  is reformulated as a function  $\underline{p}: S_1 \times \dots \times S_n \rightarrow FOR$  and imported in  $TRL$  (predicates from  $\Sigma$  are imported as functions). Similarly, for all n-ary predicates  $p \in pred(DCL)$  and all sorts  $S_1, \dots, S_n \in sort(DCL)$  where  $p: S_1 \times \dots \times S_n$ , predicate  $p$  is reformulated as a function  $\underline{p}: S_1 \times \dots \times S_n \rightarrow DCFOR$  and imported in  $TRL$  (predicates from  $DCL$  are imported as functions). Finally, logical connectives that operate on  $\Sigma$  and  $DCL$  are imported as functions, i.e.

1.  $\underline{\wedge}: FOR \times FOR \rightarrow FOR$
2.  $\underline{\vee}: FOR \times FOR \rightarrow FOR$
3.  $\underline{\rightarrow}: FOR \rightarrow FOR$

Some specific functions for the trace-requirement language that are not imported from  $\Sigma$  or  $DCL$  are as follows:

4.  $Input: SP_{Comp} \rightarrow SP_{Input}$
5.  $Output: SP_{Comp} \rightarrow SP_{Output}$
6.  $state\_r: \Gamma \times T \times SP_{Interface} \rightarrow M_R$
7.  $state\_c: \Gamma \times T \rightarrow M_C$

Given the functions as defined above, the terms of  $TRL$  are defined in usual way. The predicates mentioned in table 6 represent relations between system states and their information contents (*holds\_info*), and system states and their design configurations (*holds\_struct*).

**Table 6.** Predicates of  $TRL$

Predicate	Description
$<: T \times T$	time ordering relation
$holds\_info\_r: M_R \times FOR \times SIGN$	formula holds in state
$holds\_info\_c: M_C \times FOR \times SP_{Interface} \times SIGN$	formula holds in state of component
$holds\_struct\_r: M_R \times DEFOR \times SIGN$	design formula holds in state
$holds\_struct\_c: M_C \times DEFOR \times SP \times SIGN$	design formula holds in state of component

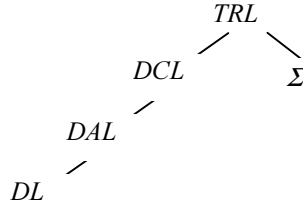
Formulae of *TRL* can now be defined as follows:

1. If  $P : S_1 \times \dots \times S_n$  is a  $n$ -ary predicate and  $t_i$  is a term of sort  $S_i$  ( $i=1, \dots, n$ ), then  $P(t_1, \dots, t_n)$  is an expression of the requirement language.
2. If  $E_1$  and  $E_2$  are expressions and  $x:S$  is a variable of sort  $S$ , then  $E_1 \wedge E_2$ ,  $E_1 \Rightarrow E_2$ ,  $\neg E_1$ ,  $\forall x:S E_1$ ,  $\exists x:S E_1$  are formulae of the requirement language.

Let  $\delta$  be a design formula of sort *DEFOR*,  $\text{agen}_{\text{MA}}$  be the Maintenance agent having sort  $\text{SP}_{\text{Comp}}$ ,  $\text{SYS}$  be the system itself, then the following *TRL* formula states that in all system traces and at any time point if the output of the maintenance agent is a communication formula expressing a creation action (i.e. *to\_be\_performed(add( $\delta$ ))*), then at some time later the system is modified according to the creation action:

$$\forall \mathcal{G}:T \exists t:T \\ [ \text{holds\_info\_c}(\text{state\_c}(\mathcal{G}:T, t:T), \text{to\_be\_performed}(\text{add}(\delta)), \text{output}(\text{agen}_{\text{MA}}), \text{true}) \Rightarrow \\ \exists t':T [ t':T > t:T \wedge \text{holds\_struct\_c}(\text{state\_c}(\mathcal{G}:T, t':T), \delta, \text{SYS}, \text{true}) ] ]$$

In the following, we will abbreviate the quantified expression “ $\exists t':T t':T > t:T \wedge \dots$ ” and write “ $\exists t':T > t:T \wedge \dots$ ” instead. Note that there is an ordering relation between defined languages (*DL*, *DAL*, *DCL*, *TRL*, and a domain specific signature  $\Sigma$ ) according to which one language imports terms and formulae from another language. This import ordering relation is illustrated in Figure 2.



**Fig. 2.** The import relation between languages

## 2.5 On Semantics

The semantical relation between language expressions, models with respect to that language and the real world containing computer systems is twofold. The part of a model (at a certain time point) representing the information state of the system corresponds to the actual information state of the computer system at that time point. Also, the part of the model at a certain time point that represents the structure of the system corresponds to the actual structure of that system at that time point.

## 3 Requirements for an Example Scenario

In this section, we use the trace requirement language *TRL* to express a number of requirements that specify relevant properties concerning both configuration as well as

information behaviour of agent-mediated systems. In order to motivate these requirements, we introduce a scenario where system configuration is modified.

### 3.1 Example Scenario

Consider a scenario for a system consisting of three components named UA (User Agent), MA (Maintenance Agent), and EW (Environment). At a certain time point, the User Agent may need a Personal Assistant agent and requests that to the Maintenance Agent. The Maintenance Agent generates an action (to be executed in the environment) to create a Personal Assistant agent for the User Agent. After the Personal Assistant agent is created (now the system consists of four components), the User Agent can communicate with its Personal Assistant agent and require certain information to be provided by the Personal Assistant agent. This scenario can be described as the following sequence of pairs indicating the system configuration and the system information states, respectively:

1. System structure consists of UA, MA and EW,  
UA (internally) identifies the need for a personal assistance.
2. System structure consists of UA, MA and EW,  
UA generates a request on its output for personal assistance.
3. System structure consists of UA, MA and EW,  
MA has the user service-request for personal assistance on its input.
4. System structure consists of UA, MA and EW,  
MA generates to\_be\_performed(add(PA)) on its output.
5. System structure consists of UA, MA and EW  
EW has to\_be\_performed(add(PA)) on its input
6. System structure consists of UA, MA, EW and PA  
EW has E (the effect of add(PA)) on its output
7. System structure consists of UA, MA, EW and PA  
UA has information-request on its output
8. System structure consists of UA, MA, EW and PA  
PA has the information request on its input
9. System structure consists of UA, MA, EW and PA  
PA has an answer to the information-request on its output
10. System structure consists of UA, MA, EW and PA  
UA has the answer on its input

Given the above sequence of system states, we distinguish the following types of operations that transfer one system state into another:

- a. Communication initiation by UA
- b. Communication event between UA and MA
- c. Action initiation by MA
- d. (World) interaction event between MA and EW
- e. Action execution by EW (resulting in creation of PA and generating E on its output)
- f. Answer generation by PA

### 3.2 Relevant properties

In this section, the trace requirement language is used to specify and express some important properties of the system in the scenario mentioned above. These properties are distinguished into three classes called *global*, *basic* and *semantic properties*. The global properties concern the behaviour of the system as a whole, the basic properties concern the behaviour of the system parts, and the semantic properties are the assumed generic properties for the type of system.

#### 3.2.1 Global Properties

Two important global properties for the example mentioned above are specified as the following requirements.

**GR1 (UA-EW Impact):** If at time point  $t$  the agent UA generates a service request  $Q$  to MA on its output, and there exists a design description  $E$  such that  $E$  would realize  $Q$ , then a time point  $t' > t$  exists such that EW has  $E$  on its output.

$$\begin{aligned} & \forall \mathcal{G}:T, t:T, Q:SLTERM, E:DEFOR \\ & [ \text{holds\_info\_r}(\text{state\_r}(\mathcal{G}:T, t:T, \text{output}(UA)), \text{communication\_from\_to}(Q:SLTERM, UA, MA), \text{true}) \\ & \wedge \text{structure\_realises\_service}(E:DEFOR, Q:SLTERM) \quad \Rightarrow \\ & \exists t':T > t:T \quad [ \text{holds\_info\_r}(\text{state\_r}(\mathcal{G}:T, t':T, \text{output}(EW)), E:DEFOR, \text{true}) ] \end{aligned}$$

Note that SLTERM is an assumed sort that refers to the service language terms by which the user denotes the kind of service (s)he is interested in. Examples that illustrate SLTERM expressions are “request(personal-assistant)” and “request(car-brokering-system)”.

**GR2 (Creation Successfulness):** If at time point  $t$  the agent UA generates a service request  $Q$  (e.g., for having personal assistance) to MA on its output, and  $R$  is the behaviour required for service request  $Q$ , then a time point  $t' > t$  exists such that the system configuration contains the necessary structure (e.g., *pers\_ass*) and the system shows behaviour  $R$ .

$$\begin{aligned} & \forall \mathcal{G}:T, t:T, Q:SLTERM \\ & [ \text{holds\_info\_r}(\text{state\_r}(\mathcal{G}:T, t:T, \text{output}(UA)), \text{communication\_from\_to}(Q:SLTERM, UA, MA), \text{true}) \\ & \wedge \exists E:DEFOR, d1:T, d2:T \quad [ \text{structure\_realises\_service}(E:DEFOR, Q:SLTERM) \\ & \wedge \text{SB1}_R(E:DEFOR, d1:T, d2:T) \quad ] \quad \Rightarrow \\ & \exists t':T > t:T, E':DEFOR, d1:T, d2:T \\ & \quad [ \text{structure\_realises\_service}(E':DEFOR, Q:SLTERM) \\ & \quad \wedge \text{holds\_info\_r}(\text{state\_r}(\mathcal{G}:T, t':T, \text{output}(EW)), E':DEFOR, \text{true}) \\ & \quad \wedge \text{SB1}_R(E':DEFOR, d1:T, d2:T) \\ & \quad \wedge \forall t1:T, t2:T, t3:T, t4:T \\ & \quad \quad [ t2:T - t1:T \geq d1:T \wedge t1:T \leq t3:T \leq t2:T - d2:T \wedge t4:T - t3:T \geq d2:T \Rightarrow \\ & \quad \quad \quad R(\mathcal{G}:T, t3:T, t4:T, \text{pers\_ass}) \quad ] \quad ] \end{aligned}$$

The terms  $SB1_R$  and  $R$  occurring in this property are explained in Section 3.2.3.3. The term  $SB1_R$  is a scheme of properties that relate a structure  $E: \text{DEFOR}$  to behaviour  $R$ . The occurrence of  $R$  makes it a scheme.

### 3.2.2 Global Properties

The basic properties are divided in three subclasses as follows.

#### 3.2.2.1 Agent properties

An agent property refers to the behaviour of a specific agent. An important agent property for the example mentioned above is specified by the following requirement.

**AR1 (MA Action Initiation Successfulness):** If at time point  $t$  the agent  $MA$  has a service request  $Q$  (e.g. a request for a personal assistance) on its input, then a time point  $t' > t$  exists such that  $MA$  has  $\text{to\_be\_performed}(\text{add}(E))$  on its output where  $E$  is a system configuration that can satisfy the service request  $Q$ .

$$\begin{aligned} & \forall \mathcal{C}: \mathcal{I}, t: \mathcal{T}, Q: \text{SLTERM}, E: \text{DEFOR} \\ & [ \text{holds\_info\_r}(\text{state\_r}(\mathcal{C}: \mathcal{I}, t: \mathcal{T}, \text{input}(MA)), \text{communication\_from\_to}(Q: \text{SLTERM}, UA, MA), \text{true}) \\ & \wedge \text{structure\_realises\_service}(E: \text{DEFOR}, Q: \text{SLTERM}) \\ & \wedge \neg \exists E': \text{DEFOR} \\ & \quad [ \text{holds\_info\_r}(\text{state\_r}(\mathcal{C}: \mathcal{I}, t: \mathcal{T}, \text{input}(MA)), \text{observation\_result}(E': \text{DEFOR}, \text{pos}), \text{true}) \\ & \quad \wedge \text{structure\_realises\_service}(E': \text{DEFOR}, Q: \text{SLTERM}) ] \Rightarrow \\ & \exists t': \mathcal{T} > t: \mathcal{T} \\ & [ \text{holds\_info\_r}(\text{state\_r}(\mathcal{C}: \mathcal{I}, t': \mathcal{T}, \text{output}(MA)), \text{to\_be\_performed}(\text{add}(E: \text{DEFOR})), \text{true}) \\ & \wedge \forall E'': \text{DEFOR} \\ & \quad [ \text{structure\_realises\_service}(E'': \text{DEFOR}, Q: \text{SLTERM}) \\ & \quad \wedge \text{holds\_info\_r}(\text{state\_r}(\mathcal{C}: \mathcal{I}, t': \mathcal{T}, \text{output}(MA)), \text{to\_be\_performed}(\text{add}(E'': \text{DEFOR})), \text{true}) ] \\ & \quad \Rightarrow \text{equal}(E: \text{DEFOR}, E'': \text{DEFOR}) ] \\ & ] \\ & ] \end{aligned}$$

#### 3.2.2.2 World properties

The required properties of the behaviour of the environment component are specified as follows.

**ER1 (EW Action Execution Successfulness):** If at time point  $t$  the world component  $EW$  has  $\text{to\_be\_performed}(\alpha)$  on its input, and  $\beta$  is the effect of performing  $\alpha$ , then a time point  $t' > t$  exists such that  $EW$  has  $\beta$  on its output.

$$\begin{aligned}
& \forall \mathcal{G}: \Gamma, t: T, \alpha: \text{ACTION}, \beta: \text{DEFOR}, \\
& [ \text{holds\_info\_r}(\text{state\_r}(\mathcal{G}: \Gamma, t: T, \text{input}(\text{EW})), \text{to\_be\_performed}(\alpha: \text{ACTION}), \text{true}) \Rightarrow \\
& \exists t': T > t: T \\
& [ \text{holds\_info\_r}(\text{state\_r}(\mathcal{G}: \Gamma, t': T, \text{output}(\text{EW})), \beta: \text{DEFOR}, \text{true}) \\
& \wedge \text{is\_effect\_of}(\beta: \text{DEFOR}, \alpha: \text{ACTION}) ] \\
& ]
\end{aligned}$$

Where the predicate  $\text{is\_effect\_of}(\beta, \alpha)$  is defined as follows:

$$\begin{aligned}
& \forall x: \text{DEFOR}^+ \quad \text{is\_effect\_of}(x, \text{add}(x)) \\
& \forall x: \text{DEAT} \quad \text{is\_effect\_of}(\neg x, \text{delete}(x))
\end{aligned}$$

### 3.2.2.3 World properties

The transfer properties specify that information transfer between agents takes place in a proper manner. Two important transfer properties are specified as the following requirements.

**TR1 (Transfer UA-MA):** If at time point  $t$  the agent UA generates a request for MA on its output, then a time point  $t' > t$  exists such that MA has this request on its input.

$$\begin{aligned}
& \forall \mathcal{G}: \Gamma, t: T, \varphi: \text{FOR} \\
& [ \text{holds\_info\_r}(\text{state\_r}(\mathcal{G}: \Gamma, t: T, \text{output}(\text{UA})), \text{communication\_from\_to}(\varphi: \text{FOR}, \text{UA}, \text{MA}), \text{true}) \Rightarrow \\
& \exists t': T > t: T \\
& [ \text{holds\_info\_r}(\text{state\_r}(\mathcal{G}: \Gamma, t': T, \text{input}(\text{MA})), \text{communication\_from\_to}(\varphi: \text{FOR}, \text{UA}, \text{MA}), \text{true}) ] \\
& ]
\end{aligned}$$

**TR2 (Transfer MA-EW):** If at time point  $t$  the agent MA generates  $\text{to\_be\_performed}(\alpha)$  for EW on its output (Note that  $\alpha$  is of type ACTION such that it is either  $\text{add}(E)$  or  $\text{delete}(E)$ ), then a time point  $t' > t$  exists such that EW has  $\text{to\_be\_performed}(\alpha)$  on its input.

$$\begin{aligned}
& \forall \mathcal{G}: \Gamma, t: T, \alpha: \text{ACTION} \\
& [ \text{holds\_info\_r}(\text{state\_r}(\mathcal{G}: \Gamma, t: T, \text{output}(\text{MA})), \text{to\_be\_performed}(\alpha: \text{ACTION}), \text{true}) \Rightarrow \\
& \exists t': T > t: T \\
& [ \text{holds\_info\_r}(\text{state\_r}(\mathcal{G}: \Gamma, t': T, \text{input}(\text{EW})), \text{to\_be\_performed}(\alpha: \text{ACTION}), \text{true}) ] \\
& ]
\end{aligned}$$

### 3.2.3 Semantic Properties

The semantic properties specify the assumed generic properties of the system. The following semantic properties are distinguished.

### 3.2.3.1 Semantic Properties

This property guarantees that if a system description holds in (the informational state of) the environment, then the actual system configuration (its structural state) is indeed described by that description.

**RA1:** If at time point  $t$  in trace  $\mathcal{G}$  the world component EW has description E on its output, then at time point  $t$  in trace  $\mathcal{G}$  the system has structure E (i.e. the system description Y is true).

$$\begin{aligned} & \forall M : \Gamma, t: T, E: \text{DEFOR} \\ & [ \text{holds\_info\_r}(\text{state\_r}(\mathcal{G}: \Gamma, t: T, \text{output}(\text{EW})), E: \text{DEFOR}, \text{true}) \quad \Rightarrow \\ & \quad \text{holds\_struct\_c}(\text{state\_c}(\mathcal{G}: \Gamma, t: T), E: \text{DEFOR}, \text{true}) ] \end{aligned}$$

### 3.2.3.2 Coherence property

The coherence properties specify a relation between the informational and structural states of the actual system at any point in time. Two important properties that guarantee the coherency of the system are as follows.

**CA1:** At any time  $t$  in any trace  $\mathcal{G}$  if a formula has a truth-value in the informational state for a specific system part, then this system part actually is part of the system structure (in the structural state).

$$\begin{aligned} & \forall M : \Gamma, t: T, C: \text{SP}, F: \text{IFOR}, E: \text{DEFOR}, \text{tv}: \text{TV} \\ & [ \text{holds\_info\_r}(\text{state\_r}(\mathcal{G}: \Gamma, t: T, \text{interface}(C: \text{SP})), F: \text{IFOR}, \text{tv}: \text{TV}) \quad \Rightarrow \\ & \quad \exists \Sigma: \text{SIG} \\ & \quad [ \text{olds\_struct\_r}(\text{state\_r}(\mathcal{G}: \Gamma, t: T, \text{system}), \text{exists\_comp}(C: \text{SP}), \text{true}) \\ & \quad \wedge \text{holds\_struct\_r}(\text{state\_r}(\mathcal{G}: \Gamma, t: T, \text{system}), \text{has\_interface\_signature}(\Sigma: \text{SIG}, C: \text{SP}), \text{true}) \\ & \quad \wedge \text{holds\_struct\_r}(\text{state\_r}(\mathcal{G}: \Gamma?, t: T, \text{system}), \text{is\_formula\_of}(F: \text{IFOR}, \Sigma: \text{SIG}), \text{true}) ] \\ & ] \end{aligned}$$

**CA2:** At any time  $t$  in any trace  $\mathcal{G}$  if the system has a certain structure (in its structural state), then the formulae that can be evaluated in this system structure have truth-values (in the system's informational state).

$$\begin{aligned} & \forall \mathcal{G}: \Gamma, t: T, C: \text{SP}, F: \text{IFOR}, E: \text{DEFOR}, \exists \Sigma: \text{SIG} \\ & [ [ \text{holds\_struct\_r}(\text{state\_r}(\mathcal{G}: \Gamma?, t: T, \text{system}), \text{exists\_comp}(C: \text{SP}), \text{true}) \\ & \quad \wedge \text{holds\_struct\_r}(\text{state\_r}(\mathcal{G}: \Gamma?, t: T, \text{system}), \text{has\_interface\_signature}(\Sigma: \text{SIG}, C: \text{SP}), \text{true}) \\ & \quad \wedge \text{holds\_struct\_r}(\text{state\_r}(\mathcal{G}: \Gamma?, t: T, \text{system}), \text{is\_formula\_of}(F: \text{IFOR}, \Sigma: \text{SIG}), \text{true}) ] \\ & \quad \Rightarrow \\ & \quad \exists \text{tv}: \text{TV} \text{ holds\_info\_r}(\text{state\_r}(\mathcal{G}: \Gamma, t: T, \text{interface}(C: \text{SP})), F: \text{IFOR}, \text{tv}: \text{TV}) ] \end{aligned}$$

### 3.2.3.3 Structure-Behaviour properties

In this section (required) behaviour is related to structure properties and service requests. In the following  $R(\mathcal{G}; T, t1: T, t2: T, C: SP)$  stands for a certain behavioural requirement. As an example,  $pers\_ass\_req(\mathcal{G}; T, t1: T, t2: T, C: SP)$  denotes the following requirement; here INFO is a sort for the content of requests and answers:

$$\begin{aligned} & \forall A: INFO [ holds\_info\_r(state\_r(\mathcal{G}; T, t1: T, input(C: SP)), request(A: INFO), true) \Rightarrow \\ & \exists t: T, \exists B: INFO [ t1: T \leq t: T \leq t1: T + t2: T \wedge \\ & holds\_info\_r(state\_r(\mathcal{G}; T, t: T, output(C: SP)), answer\_for(B: INFO, A: INFO), true) ] ] \end{aligned}$$

By defining such abbreviations for the requirements of interest for the system in question a powerful scheme of structure-behaviour properties can be formulated:  $SB1_R$  and  $SB2_R$ .

Let  $SB1_R ( E: DEFOR, d1: T, d2: T )$  denote the following scheme of structure-behaviour properties: If between time points  $t1$  and  $t2$  the system has structure  $E: DEFOR$ , then  $R$  is true between  $t1$  and  $t2 - d2$  for all periods of sufficient length ( $d1$  minimum):

$$\begin{aligned} & \forall \mathcal{G}; T, \forall t1: T, t2: T, t3: T, t4: T \\ & [ [ t2: T - t1: T \geq d1: T \\ & \wedge \forall t \in [t1: T, t2: T] holds\_struct\_c(state\_c(\mathcal{G}; T, t), E: DEFOR, true) \\ & \wedge t1: T \leq t3: T \leq t2: T - d2: T \wedge t4: T - t3: T \geq d2: T ] \Rightarrow \\ & R(\mathcal{G}; T, t3: T, t4: T, pers\_ass) ] \end{aligned}$$

In the next structure-behaviour scheme of properties  $SB2_R$ , the link is made between a service request  $Q$ , the behaviour  $R$  that satisfies  $Q$ , and the possible structures  $E: DEFOR$  that can realise  $R$ .  $SB2_R$  denotes the following scheme of structure-behaviour properties:

$$\begin{aligned} & \forall Q: SLTERM \\ & [ implied\_service_R(Q: SLTERM) \Rightarrow \\ & \exists E: DEFOR, d1: T, d2: T \\ & [ SB1_R(E: DEFOR, d1: T, d2: T) \\ & \wedge structure\_realises\_service(E: DEFOR, Q: SLTERM) ] \\ & ] \end{aligned}$$

## 4 System Evaluation

In order to evaluate the system behaviour, we need to verify that the system has the required properties. In Section 3, we have introduced certain properties for the proposed scenario. However, verifying such properties may be quite complex. Therefore, we may define some intermediate properties by means of which global properties can

be proved easily and more efficiently, and which, by themselves, can be derived from basic properties.

#### 4.1 Intermediate Properties

Intermediate properties can be derived from basic properties. However, it is useful to formulate them explicitly. These properties specify the output/output relations between different system components. Two important intermediate properties are as follows.

**IR1 (UA-MA Interaction):** If at time point  $t$  the agent UA generates a service request  $Q$  for MA on its output, then a time point  $t' > t$  exists such that MA has to\_be\_performed(add(E)) related to this request on its output.

$$\begin{aligned} & \forall \mathcal{M}: \mathcal{L}, t: T, Q: \text{SLTERM}, E: \text{DEFOR} \\ & [ [\text{holds\_info\_r}(\text{state\_r}(\mathcal{M}: \mathcal{L}, t: T, \text{output}(\text{UA})), \text{communication\_from\_to}(Q: \text{SLTERM}, \text{UA}, \text{MA}), \text{true}) \\ & \quad \wedge \text{structure\_realises\_service}(E: \text{DEFOR}, Q: \text{SLTERM}) \\ & \quad \wedge \neg \exists E': \text{DEFOR} \\ & \quad \quad [\text{holds\_info\_r}(\text{state\_r}(\mathcal{M}: \mathcal{L}, t: T, \text{output}(\text{EW})), \text{observation\_result}(E': \text{DEFOR}, \text{pos}), \text{true}) \\ & \quad \quad \wedge \text{structure\_realises\_service}(E': \text{DEFOR}, Q: \text{SLTERM}) ] ] \Rightarrow \\ & \exists t': T > t: T \\ & \quad [ \text{holds\_info\_r}(\text{state\_r}(\mathcal{M}: \mathcal{L}, t': T, \text{output}(\text{MA})), \text{to\_be\_performed}(\text{add}(E: \text{DEFOR})), \text{true}) \\ & \quad \quad \wedge \forall E'': \text{DEFOR} \\ & \quad \quad \quad [ \text{structure\_realises\_service}(E'': \text{DEFOR}, Q: \text{SLTERM}) \\ & \quad \quad \quad \quad \wedge \text{holds\_info\_r}(\text{state\_r}(\mathcal{M}: \mathcal{L}, t': T, \text{output}(\text{MA})), \text{to\_be\_performed}(\text{add}(E'': \text{DEFOR})), \text{true}) \\ & \quad \quad \quad \quad \Rightarrow \text{equal}(E: \text{DEFOR}, E'': \text{DEFOR}) \\ & \quad \quad \quad ] \\ & \quad \quad ] \\ & \quad ] \\ & ] \end{aligned}$$

**IR2 (MA-EW Interaction):** If at time point  $t$  MA has an action (e.g., to\_be\_performed(add(E)) or to\_be\_performed(delete(E))) on its output, then a time point  $t' > t$  exists such that EW has the effect of the action (e.g., E resp. not E) on its output.

$$\begin{aligned} & \forall \mathcal{M}: \mathcal{L}, t: T, \alpha: \text{ACTION}, \beta: \text{DEFOR} \\ & [ [ \text{holds\_info\_r}(\text{state\_r}(\mathcal{M}: \mathcal{L}, t: T, \text{output}(\text{MA})), \text{to\_be\_performed}(\alpha: \text{ACTION}), \text{true}) \\ & \quad \wedge \text{is\_effect\_of}(\beta: \text{DEFOR}, \alpha: \text{ACTION}) ] \Rightarrow \\ & \exists t': T > t: T \\ & \quad \text{holds\_info\_r}(\text{state\_r}(\mathcal{M}: \mathcal{L}, t': T, \text{output}(\text{EW})), \beta: \text{DEFOR}, \text{true}) ] \end{aligned}$$

#### 4.2 The Use of Proof Patterns

The following logical relationships hold between the different properties.

1. Intermediate properties are implied by transfer properties and agent or world properties:

$$\text{TR1} \ \& \ \text{AR1} \Rightarrow \text{IR1}$$

$$\text{TR2} \ \& \ \text{ER1} \Rightarrow \text{IR2}$$

2. Intermediate properties can be chained to indirect interaction properties expressing indirect impact of one component on another one:

$$\text{IR1} \ \& \ \text{IR2} \Rightarrow \text{GR1}$$

3. Indirect intermediate properties imply the global properties, assuming Representation and Coherence properties and Structure-Behaviour relationships:

$$\text{GR1} \ \& \ \text{RA1} \ \& \ \text{CA1} \ \& \ \text{SB2} \Rightarrow \text{GR2}$$

### 4.3 Checking Properties

Given a trace or set of traces, all of the above properties can be checked automatically. To this end software environment has been developed in Prolog. If the property GR2 is checked, the outcome can be one of the following

- (1) Indeed GR2 satisfied, or
- (2) GR2 is dissatisfied; in this case, given the logical relationships defined by the proof patterns, at least some of the basic properties are also dissatisfied; which one(s) can also be found by running the checking software for all of the basic properties; the outcome of this check points to where the problem originates.

## 5 Discussion

The requirements specification language introduced in this paper can be used in a number of ways. First, it allows for specification of requirements on the system structure over time. Most other requirement languages (e.g., [8], [4]) only allow for specification of informational system states, for a given, fixed system structure. In our language it is possible both to refer to the structural state of the system and the informational state of it.

A second use is that a broad class of behavioural properties can be specified of the system or of specific parts of the system, for example agents. Not only can, e.g., reactivity and pro-activeness properties be specified, but also properties expressing adaptive behaviour, such as 'exercise improves skill', which are relative to (comparing two alternatives for) the history can be expressed in this language (in standard forms of temporal logic different alternative histories cannot be compared).

A third and more sophisticated use is to specify requirements on the dynamics of the process of modification of the system structure over time, for example, as initiated and performed by the system (e.g., one of its agents) itself. Here requirements on, for example, agent behaviour and the dynamics of the system structure and their relation-

ships can be specified. For example it can be expressed whether a system modification initiation by one of the agents occurs and whether it is successful.

Requirements can be specified at different levels of aggregation. For example, a requirement for the overall system can be refined into requirements of different parts of the system, i.e., requirements on specific agents and on specific interactions between agents, which, together, imply the global requirement.

For all different types of requirements discussed, for a given set of traces the requirements can be verified automatically. By specifying the refinement of a requirement for the overall system, it is possible to perform diagnosis of malfunctioning of the system. If the overall requirement fails on a given trace, then subsequently, all refined requirements for the parts of the system can be verified against that trace: the cause of the malfunctioning can be attributed to the part(s) of the system for which the refined requirement(s) fail(s).

## References

1. Brazier, F.M.T., Jonker, C.M., and Treur, J., Principles of Compositional Multi-agent System Development. In: J. Cuenca (ed.), *Proc. of the 15<sup>th</sup> IFIP World Computer Congress, WCC'98, Conference on Information Technology and Knowledge Systems, IT&KNOWS'98*, 1998, pp. 347-360. To be published by IOS Press.
2. Davis, A. M. (1993). *Software requirements: Objects, Functions, and States*, Prentice Hall, New Jersey.
3. Dardenne, A., Lamsweerde, A. van, and Fickas, S. (1993). Goal-directed Requirements Acquisition. *Science in Computer Programming*, vol. 20, pp. 3-50.
4. Darimont, R., and Lamsweerde, A. van (1996). Formal Refinement Patterns for Goal-Driven Requirements Elaboration. *Proc. of the Fourth ACM Symposium on the Foundation of Software Engineering (FSE4)*, pp. 179-190.
5. Dubois, E., Du Bois, P., and Zeippen, J.M. (1995). A Formal Requirements Engineering Method for Real-Time, Concurrent, and Distributed Systems. In: *Proceedings of the Real-Time Systems Conference, RTS'95*.
6. Dubois, E., Yu, E., Petit, M. (1998). From Early to Late Formal Requirements. In: *Proc. IWSSD '98*. IEEE Computer Society Press.
7. Engelfriet, J., Jonker, C.M. and Treur, J., (1999). Compositional Verification of Multi-Agent Systems in Temporal Multi-Epistemic Logic. In: J.P. Mueller, M.P. Singh, A.S. Rao (eds.), *Intelligent Agents V, Proc. of the Fifth International Workshop on Agent Theories, Architectures and Languages, ATAL'98*. Lecture Notes in AI, vol. 1555, Springer Verlag, 1999, pp. 177-194. Extended version in: *Journal of Logic, Language and Information*, in press.
8. Fisher, M., Wooldridge, M. (1997) On the Formal Specification and Verification of Multi-Agent Systems. *International Journal of Cooperative Information Systems*, M. Huhns, M. Singh, (eds.), special issue on Formal Methods in Cooperative Information Systems: Multi-Agent Systems, vol. 6, pp. 67-94.
9. Herlea, D.E., Jonker, C.M., Treur, J., and Wijngaards, N.J.E., (1999). Specification of Behavioural Requirements within Compositional Multi-Agent System Design. In: F.J. Garijo, M. Boman (eds.), *Multi-Agent System Engineering, Proceedings of the 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW'99*. Lecture Notes in AI, vol. 1647, Springer Verlag, Berlin, 1999, pp. 8-27.

10. Jonker, C.M. and Treur, J. (1998). Compositional Verification of Multi-Agent Systems: a Formal Analysis of Pro-activeness and Reactiveness. In: W.P. de Roever, H. Langmaack, A. Pnueli (eds.), *Proceedings of the International Workshop on Compositionality, COMPOS'97*. Lecture Notes in Computer Science, **1536**, Springer Verlag, 1998, pp. 350-380. Extended version in: *International Journal of Cooperative Information Systems*, in press.
11. Kontonya, G., and Sommerville, I. (1998). *Requirements Engineering: Processes and Techniques*. John Wiley and Sons, New York.
12. Lamsweerde, A. van, Darimont, R., and Letier, E. (1998). Managing Conflicts in Goal-Driven Requirements Engineering. *IEEE Transactions on Software Engineering*, Special Issue on Managing Inconsistency in Software Engineering.
13. Manna, Z., and Pnueli, A.. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1995.
14. Sommerville, I., and Sawyer P. (1997). *Requirements Engineering: a good practice guide*. John Wiley & Sons, Chicester, England.