A Requirement Specification Language for Configuration Dynamics of Multiagent Systems

Mehdi Dastani,[‡] Catholijn M. Jonker,[†] Jan Treur[*]
Vrije Universiteit Amsterdam, Department of Artificial Intelligence, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands

In agent-mediated applications, the system configuration can change because of the creation and the deletion of agents. The behavior of such systems on the one hand depends on the dynamics of the system configuration; on the other hand, behavior of such a system consists of the information dynamics of the system. We discuss configuration and information dynamics of agent-mediated systems and define a requirement language to express properties of those dynamics. A prototypical scenario for an agent-mediated system is discussed and some important requirements for this system are specified. It is shown how these properties can be verified automatically to evaluate system behavior. © 2004 Wiley Periodicals, Inc.

1. INTRODUCTION

Requirements Engineering is a well-studied field of research within software engineering; e.g., see Refs. 1–3. Requirements specify the required properties of a system, which include the functions of the system, structure of the system, and static and dynamic properties. Recently requirements engineering for distributed and agent systems has been studied in some depth, e.g., see Refs. 4 and 5. For agent-based systems, the dynamics or behavior 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 requirement specification languages for dynamics 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 requirement specification language. A simple form of requirements for dynamics

[*]Author to whom all correspondence should be addressed: e-mail: treur@cs.vu.nl.
[†]e-mail: jonker@cs.vu.nl.
[‡]Current address: Utrecht University, Institute of Information and Computing Sciences, Padualaan 14, 3584 CH Utrecht, The Netherlands. e-mail: mehdi@cs.vu.nl.

INTERNATIONAL JOURNAL OF INTELLIGENT SYSTEMS, VOL. 19, 277–300 (2004)
 Published online in Wiley InterScience (www.interscience.wiley.com). • DOI 10.1002/int.10171

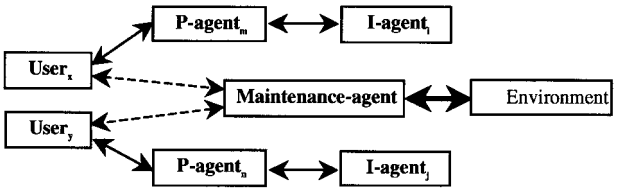

Figure 1. The generic architecture of the agent-mediating systems.

(close to functional requirements) expresses reactive types of behavior. However, combinations of proactive and reactive behavior can require more complicated requirement expressions. Most types of behavior can be expressed using rather standard forms of temporal logic.[6–8] Additional 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 behavior of this type are relative adaptive behavior (e.g., "exercise improves skill"), in which two different possible histories have to be compared, and self-modifying behavior, e.g., the behavior of one of the agents (e.g., initiating a creation action of an additional agent), leads to a different system configuration.

This article addresses specifications of requirements for the self-modifying type of dynamics. As illustrations of this type of dynamics, consider self-modifying extensions of agent-mediating systems such as brokering systems, matchmaking systems, and search engines. In these applications, a user interacts with the system and receives its personal assistant (PA), which in turn interacts with available intermediate and task-specific agents to perform the user task. The self-modification of these systems lies in the idea that agents (PA and task-specific agents) are created and removed from the system according to circumstances.

An abstract multiagent architecture for such self-modifying mediating systems is illustrated schematically in Figure 1. According to this architecture, self-modifying mediating systems consist of different types of components or agents, including a central maintenance agent (MA), an environment component, PA agents (P-agent), and intermediate task-specific agents (I-agent). The MA is responsible for initial interactions with users and the overall configuration of the system. For example, a user who wants to buy a car communicates with the MA and requires a car brokering system. With respect to the received request, the MA specifies a car brokering system consisting of a car broker agent (which may already exist) and a PA agent for the user (which may already exist) in such a way that the car broker agent and PA agent can interact to respond adequately to the request. The MA can conclude that the configuration of the system needs to be modified. If so, then a specification of necessary changes to the system is communicated to the environment component to modify the system configuration. The environment component represents all ingredients of the system except itself and is responsible for the realization of system modifications. The environment component is assumed to have a causal relation with the actual environment of the agent-mediating system in the sense that if it includes the representation of a component, then the component exists in the actual environment, and if there is a

component in the actual environment, then it is represented in the environment component. Moreover, the environment component is assumed to include a subcomponent, which translates modification plans, which it receives from the MA, to an appropriate representation of system configuration. By the causal relation between the environment component and the actual environment, the system configuration is guaranteed to be modified appropriately. The user can now delegate its car-buying task to its PA agent, which in turn can interact with the car broker agent to achieve the delegated user task. The user can decide to log-off from the broking system, in which case the PA agent and the car broker agent either can be deleted from the system or set idle.

The MA and the environment processes form the basic components of the system. They will be created at startup of the system and remain until the system terminates. An essential property of such self-modifying agent-mediating systems is the dynamic nature of their configuration: new agents are created and removed from the system. The correct behavior of the system depends on its configuration during its execution. For this reason, it is important to specify configuration properties of such systems during their execution. The specification and verification of such configuration properties is the focus of this study.

The article is organized as follows. In Section 2 and 3 state and design languages are introduced. In Section 4 and 5 a requirement language is introduced that can express, besides functional and informational requirements, those requirements that are concerned with the system configuration during its execution. In Section 6, this requirement language is used to identify a number of requirement types concerning the configuration of agent-mediated systems. A scenario for an agent-mediating system will motivate these requirements. Section 7 discusses possibilities for the verification of the requirements proposed in Section 6, and Section 8 concludes the study.

2. SYSTEM SEMANTICS

The aim of this study is to specify and verify configuration properties of self-modifying agent systems. This requires that the behavior of a system is formally understood in general, before specializing to self-modifying systems. In this section, first attention is given to state languages, i.e., languages with which different states of systems can be expressed. In the second part of this section, the semantics of systems are defined in terms of information states of the system.

2.1. State Languages

In general, a state of a system can be considered as a valuation of the language elements (i.e., an assignment of truth values) that are used within or between components of the system. Within agent systems the interaction with the external world (EW) and other agents plays a central role. Agents reason what communications, actions, and observations should be performed based on information obtained through observation and communication. Agents further determine how to interpret observation results and information received through communication.

In general, these aspects can be represented by statements expressed in an order-sorted predicate language. The sorts of this language relevant for the mentioned aspects are the following:

ACTION
INFORMATION_ELEMENT
SIGN
AGENT

The objects belonging to the different sorts change from system to system. The relevant predicates belonging to the language are the following:

to_be_performed:	ACTION
to_be_observed:	INFORMATION_ELEMENT
observation_result:	INFORMATION_ELEMENT $\times$ SIGN
communication_from_to:	INFORMATION_ELEMENT $\times$ AGENT $\times$ AGENT

Using this language, it is possible to represent statements like communicated_by(request(personal-assistant), pos, user("Johnson")). In this expression, the application-specific parts are terms of sort INFORMATION_ELEMENT and AGENT. This expression can be generated in the mediating system example if the user identified as user("Johnson") initiates a request for a PA. Another example expression particular to the example system is the representation of the decision to add a PA component called pers_ass: to_be_performed(add(exists_comp-(pers_ass))). Of course, for other applications, the content of the actions, information elements, and agents are appropriate for those applications.

2.2. Information States

An understanding of system behavior can be obtained by describing the semantics of the system; see, e.g., Ref. 9. The semantics can be studied from a *static* and a *dynamic perspective.* The static semantics of a system specification can be described by the information states of the system. In principle, an *information state* I of a (part of a) system S (e.g., the overall system or an input or output interface of an agent) is an assignment of truth values {true, false, unknown} to the set of ground atoms describing the information within S. The language elements used to form the ground atoms can be based on the state language of the system as addressed in the previous section. Information states reflect the structure of the system. Before a formal definition can be given, first, the structure of the system has to be understood.

A compositional system, seen as a composed component, consists of a number of components. A primitive component is a component that is not composed of other components. The behavior of a compositional system is determined by the specification of the composition relation and control specified over the components. The composition relation refers to the possibilities of information transfer between components. The control refers to activation sequences of component

activations and information transfer occasions, and, therefore, to the run-time dynamics of the system.

A component receives information as input and by its own processing generates information in its output interface. At each moment, the information state of a component is defined by the information it has explicitly available in its interfaces (i.e., the input and output interfaces) and the information it has explicitly available internally at that moment.

An information state of a system is defined inductively as follows.

DEFINITION. *Information State*: *For any component C*, input(*C*) *denotes the input interface of C*, output(*C*) *denotes the output interface of C*, *and* sc(*C*) *denotes the set of subcomponents of C*. *For any D that is either a component or an interface of a component*, AT(*D*) *denotes the set of ground atoms of the language used to describe the information in D*. *Let* TV *be the set of truth values* {*false*, *unknown*, *true*}.

1. An information state of an interface D is a mapping I: AT(D) $\rightarrow$ TV. IS(D) denotes the set of all information states of interface D.
2. An information state of a primitive component C is a tuple ⟨I_input, I_output⟩, where I_input is an information state of input(C) and I_output is an information state of output(C); IS(C) denotes the set of all information states of C.
3. For component C, an interface information state is a pair ⟨I_input, I_output⟩, where I_input is an information state of input(C), and I_output is an information state of output(C). IntIS(C) denotes the set of all interface information states of C.
4. The set of all information states of a composed component C, is the set IS(C) = IntIS(C) $\times$ $\Pi_{D \in sc(C)}$IS(D). So that one information state of C is a tuple ⟨I_int, I⟩, where I_int is an interface information state of C, and I $\in$ $\Pi_{D \in sc(C)}$IS(D) is a tuple consisting of information states of the different subcomponents of C.
5. The set of all possible information states of a system S is denoted by IS(S).

For example, consider a system S consisting of two components $C1$ and $C2$. Assume that AT(Input($C1$)) = AT(output($C1$)) = $\{p1, p2\}$, AT(input($C2$)) = $\{p2\}$, and AT(output($C2$)) = $\{p3\}$. Consider the following information states:

1. $I1 \in$ IS(input($C1$)) is the information state of the input of $C1$ that is defined by $I1(p1)$ = true, $I1(P2)$ = unknown.
2. $I2 \in$ IS(output($C1$)) is the information state of the output of $C1$ that is defined by $I2(P1)$ = true, $I2(p2)$ = false.
3. $I3 \in$ IS(input($C2$)) is the information state of the input of $C2$ that is defined by $I3(p2)$ = false.
4. $I4 \in$ IS(output($C2$)) is the information state of the output of $C2$ that is defined by $I4(p3)$ = true.

Then $\langle\langle\varnothing, \varnothing\rangle, \langle\langle I1, I2\rangle, \langle I3, I4\rangle\rangle\rangle$ is an information state of S.

The foregoing definitions form a standard application of logic to represent formally reality, in this case the state of a software system or component (Figure 2). Note that the languages needed to describe the information in two (interfaces of) components can be different, but they may overlap. More details and a more extensive definition of compositional information states can be found in Ref. 9.

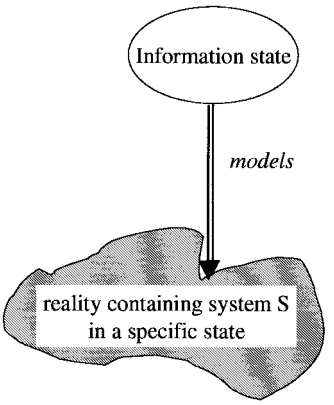

Figure 2. Information state.

The dynamic aspects of the semantics are based on evolutions of compositional information states over time (Figure 3). A *trace* γ of S is a sequence of information states $(I^t)_{t \in \mathbf{N}}$ in IS(S). Given a trace γ of S, the information state of the input interface of an agent A at time point t is denoted by state$_S(\gamma, t,$ input(A)). Analogously, state$_S(\gamma, t,$ output(A)), denotes the information state of the output interface of agent A at time point t within system S. More details on formal semantics of system specifications can be found in Refs. 9 and 10.

3. DESCRIBING DESIGNS

In the previous section, the static and dynamic semantics of systems in general has been discussed. A number of agent-specific language elements have been

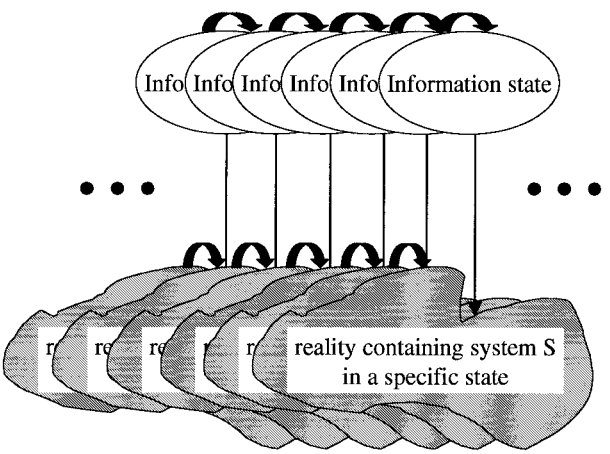

Figure 3. Traces of information states.

Table I. Important sorts in DL.

Sort	Description
SP	Sort for system parts; part names identify specific system components in the hierarchical system structure, interfaces of those components, and connections between components
SP_{comp}	Sort for system components
SP_{prim}	Sort for primitive system components
$SP_{interface}$	Sort for component interfaces (i.e., input or output)
SP_{input}	Sort for component input interfaces
SP_{output}	Sort for component output interfaces
SCON	Sort for connections between system components
SSIG	Sort for signatures used by components
SKB	Sort for knowledge bases of primitive system components

defined, but otherwise no application-specific language elements were presented. The systems central to this work are self-modifying agent systems. Before specifications can be given for such systems, an expressive and generic language needs to be introduced that allows the representation of system configurations. The language introduced here for that purpose is called design language (DL).

A characteristic of the systems studied in this article is that they have some form of compositional architecture. The reader should note that the proposed object-level language could be reformulated easily for different types of systems. A method for the design of compositional multiagent systems is the design method DESIRE; cf. Ref. 11. The language DL introduced here has been developed for the representation of compositional system designs. 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 called primitive components. These components are defined in terms of ingredients such as input and output interfaces, control loop, signatures, embedded components (for nonprimitive components), and knowledge bases (for primitive components). The configuration properties that are to be formalized concern these ingredients and their structural relations. Although an exhaustive list of all ingredients that may be used in these types of systems is beyond the scope of this study, the most important ingredients are introduced.

The design language DL is an order-sorted language that expresses the configuration of multiagent systems. The sorts in DL identify the ingredients in such systems. Although an exhaustive enumeration of all sorts is out of the scope of this study, some important sorts are mentioned in Table I.

These sorts are partially ordered: $SP_{prim} < SP_{comp} < SP$, $SP_{input} < SP_{interface} < SP$, $SP_{output} < SP_{interface}$, $SCON < SP$. For example, $SP_{prim} < SP$ states that primitive components are system parts. For all sorts S a set of constants (i.e., $\{c|c$ is a constant of sort $S\}$) and a set of variables (i.e., $\{x: S|x$ is a variable of sort $S\}$) are assumed. Also, a set of n-ary functions (i.e., $\{f^n|f^n$ is an n-ary function for any $n\}$) is assumed. For example, input: $SP \rightarrow SP$ is a function that maps components to their input parts. Given the foregoing ingredients, the terms of the DL can be defined as follows:

1. If c is a constant of sort S, then c is a term of sort S.

Table II. 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: SPprim × SKB	Primitive component has knowledge base

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

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

Based on these sorted predicates, the formulas of the DL can be defined as follows:

1. If p: $S_1 \times \cdots \times S_n$ is a n-ary predicate and t_i ($i = 1, \ldots, n$) is a term of sort S_i, then $P(t_1, \ldots, t_n)$ is a formula of DL.
2. If E_1 and E_2 are formulas of DL, then $E_1 \wedge E_2$ and $\neg E_1$ are formulas of DL.

Let sig_Σ be a signature of language Σ. The formula connected_to(agent_{PA}, agent_{IA}, from_to_3) $\wedge$ has_input_Signature(agent_{PA}, sig_Σ) is a design formula expressing that the personal assistant agent_{PA} is connected to the intermediate agent agent_{IA} by connection from_to_3 and the agent agent_{PA} uses input signature sig_Σ.

4. REASONING ABOUT SYSTEM CONFIGURATIONS

Within the scope of the example self-modifying agent-mediating system, DL expressions are used by the MA and the environment component to represent system configurations. The configuration of the system can be modified by execution of modification actions or plans that add/remove some system ingredients to/from the system. The MA and the environment component use these modification plans to modify the system configuration. Therefore, the language DL is extended to allow agents to determine design modification plans and their executions. In addition to the expressions of these configuration languages, agents need to represent knowledge about the domain of mediation. Therefore, a language, called domain language (Σ), is assumed to express the knowledge about the domain of mediation. Finally, the components in the agent-mediating system communicate expressions from DL and Σ to exchange data about the domain of

Table III. Sorts of CAL.

Sort	Description
S	$S \in \text{sort(DL)} \cup \text{sort}(\Sigma)$
DEAT	Sort for atomic formulas from DL
DEFOR	Sort for formulas from DL
DEFOR^+	Sort for positive DL formulas (atomic formulas or conjunction of positive formulas)
ΣFOR	Sort for formulas from Σ
INFORMATION_ELEMENT	Sort for formulas from Σ or DL
ACTION	Sort for actions
SIGN	Sort for truth values
AGENT	Sort for agent names

mediation, the system configuration, and modification plans. The language, called communication action language (CAL), is used to communicate these expressions. The language CAL has terms that denote the expressions of DL and Σ and uses the same predicates for communication, observation, and action determination as those that were introduced in Section 2.

The language CAL has been developed to enable the expression of plans to modify system configurations. These plans are addition or deletion commands that add/delete an ingredient to/from a system. To express these plans, terms are needed that denote system ingredients. Therefore, CAL terms will be generated for all DL terms and formulas. For this purpose, all DL sorts have been imported into CAL and the sort DEFOR has been introduced for terms that denote DL formulas. Moreover, two new subsorts, DEAT and DEFOR^+, respectively, have been introduced for terms that denote DL atomic and positive formulas. A DL-positive formula is one in which the negation symbol does not occur.

Furthermore, statements about the domain expressed in Σ also need to be available as terms in CAL. Let sort(L), func(L), and pred(L) be the sets of sorts, functions, and predicates from some language L, respectively. The sorts in CAL are presented in Table III.

Note that the following subsort relations hold: DEAT $<$ DEFOR^+ $<$ DEFOR $<$ INFORMATION_ELEMENT and ΣFOR $<$ INFORMATION_ELEMENT. For all sorts a set of constants and a set of variables are assumed. The final step to generate CAL terms for all DL expressions (DL terms and formulas) is to introduce for each DL function and predicate a CAL function. Therefore, the following CAL functions are introduced:

1. For L being either DL or Σ and all n-ary function $f \in \text{func}(L)$ and all sorts $S, S_1, \ldots, S_n \in \text{sort}(L)$, where f: $S_1 \times \cdots \times S_n \rightarrow S$, the function $\underline{f}$: $S_1 \times \cdots \times S_n \rightarrow S$ is an imported function in CAL (function f from L corresponds with function $\underline{f}$ in CAL).
2. For L being either DL or Σ and all n-ary predicates $p \in \text{pred}(L)$ and all sorts $S_1, \ldots, S_n \in \text{sort}(L)$ where p: $S_1 \times \cdots \times S_n$, predicate p is reformulated as a function $\overline{p}$: $S_1 \times \cdots \times S_n \rightarrow$ DEAT and imported in CAL (predicate name p from L corresponds to function name $\overline{p}$ in CAL).

Also, for each logical connective that connects DL formulas, a CAL function has been introduced as follows:

$$\underline{\wedge}: \text{DEFOR}^+ \times \text{DEFOR}^+ \rightarrow \text{DEFOR}^+$$

$$\underline{\wedge}: \text{DEFOR} \times \text{DEFOR} \rightarrow \text{DEFOR}$$

$$\underline{\neg}: \text{DEFOR} \rightarrow \text{DEFOR}$$

Similarly, the logical connectives that operate on Σ formulas are imported as CAL functions. The functions in CAL, which correspond with DL or Σ functions or predicates, are marked (by being underlined) in order to distinguish them from their corresponding functions or predicates from DL and Σ. Whenever there is no confusion, these underlying marks are left out. With the foregoing introductions, all terms of DL and Σ are available in CAL, and all formulas of DL and Σ are available as terms in CAL.

In addition to these functions, two specific plan-related functions have been introduced in CAL that are used in to express plans for modification of system configuration. These two plan-related functions are defined as follows:

$$\text{add}: \text{DEFOR}^+ \rightarrow \text{ACTION}$$

$$\text{delete}: \text{DEAT} \rightarrow \text{ACTION}$$

Note that the add function is not defined on negative DL formulas because they do not specify a unique system configuration. Similarly, the delete function has been defined only for atomic design expressions to avoid confusion in the semantics of the action. If the delete action would have been defined for nonatomic expressions, then what would, e.g., "delete($a \wedge b$)" mean? Would the deletion of "a" be enough, or that of "b"?

For all sorts a set of constants and a set of variables are assumed. Given these constants, variables, and functions, CAL terms are defined in the usual way. The plans (CAL terms of sort ACTION) are limited to adding and deleting a positively (resp. atomically) stated system configuration. The add function maps a positive DL formula to a plan, which when executed will result in a larger system configuration. The positive DL formula, which is the argument of the add function, specifies the system ingredients that should be added to the system.

As an example of CAL terms, consider the following expression:

$$\text{add}(\text{connected\_to}(\text{agent}_{\text{PA}}, \text{agent}_{\text{IA}}, \text{from\_to}_3)$$

$$\wedge\, \text{has\_input\_signature}(\text{agent}_{\text{PA}}, \text{sig}_{\Sigma}))$$

This expression denotes a plan to create a system consisting of agent_{PA} and agent_{IA} that are connected by connection from_to_3 and, moreover, agent_{PA} uses the input signature sig_{Σ}. The function 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.

The language CAL has the following predicates:

to_be_performed:	ACTION
to_be_observed:	INFORMATION_ELEMENT
observation_result:	INFORMATION_ELEMENT × SIGN
communication_from_to:	INFORMATION_ELEMENT × AGENT × AGENT

Based on the defined terms of the predicates, the formulas of CAL can be defined as follows:

1. If p: $S_1 \times \cdots \times S_n$ is a n-ary predicate and t_i is a term of sort S_i ($i = 1, \ldots, n$), then $p(t_1, \ldots, t_n)$ is an atomic formula of the communication action language CAL.
2. All atomic CAL formulas are CAL formulas.
3. If E, E_1 and E_2 are CAL formulas, then $\neg E$ and $E_1 \wedge E_2$ are CAL formulas.

As an example of a CAL formula, consider the following expression:

$$\text{to\_be\_performed}(\text{add}(\text{connected\_to}(\text{agent}_{\text{PA}}, \text{agent}_{\text{IA}}, \text{from\_to}_3) \wedge \text{has\_input\_signature}(\text{agent}_{\text{PA}}, \text{sig}_{\Sigma})))$$

is a design communication formula, which states that the add-action should be performed.

5. REQUIREMENT LANGUAGE

In this section, a requirement language is introduced to specify and verify requirements for self-modifying agent systems. Some examples are presented that are based on the example mediating system. This requirement language depends and builds on the language CAL used by the agents to represent and communicate information about either the domain of mediation or the system configuration.

Based on the language CAL, we define the temporal configuration requirements language (TCRL) to specify and verify the dynamic (information and configuration) behavior of agent-mediating systems. In general, requirements can be distinguished into two types: information requirements and configuration requirements. Information requirements formulate properties related to the domain knowledge and configuration properties formulate properties related to the system configuration. It is important to note that the introduction of the environment component in the example agent-mediating system and its causal relation with the actual environment make it possible to consider configuration properties as a specific type of information properties. In fact, the causal relation makes it possible to derive configuration properties of the system from its representation in the environment component.

Expressions of the languages DL and CAL are used by the components in the agent-mediating systems, while the expressions of the requirement language are used by an external observer to specify the (information and configuration) properties of the system behavior. In that sense, TCRL is a metalanguage with respect to DL and CAL.

Table IV. Sorts of TCRL.

Sort	Description
S	$S \in$ sort(CAL)
FOR	Sort for formulas from CAL
M_R	Sort for restricted states
M_C	Sort for complete states
T	Sort for time
Γ	Sort for trace

The requirement language TCRL is defined to express properties or requirements of agent-mediated systems concerning either their configuration behavior or their information behavior. Configuration behavior addresses questions such as "what will be the system configuration if the MA communicates a CAL expression to the environment component?" Information behavior addresses questions such as "what will be the system information state if a component sends an expression to another component?"

Thus, TCRL has been designed to allow the formulation of expressions concerning the configuration and information behavior of a system through time. For this reason, TCRL contains terms that denote system states, system traces, and time. System states and traces are introduced in Section 2. In light of the requirements that need to be formulated, a system state addresses both the information content of the system and the structural configuration state of the system. Moreover, TCRL contains predicates to represent relations between those terms. Thus, TCRL expressions can be used to specify properties of the configuration and information behavior of a system through time.

For any order-sorted predicate language L, sorts, functions, and predicates from L are denoted by sort(L), func(L), and pred(L), respectively. The sorts in TCRL are mentioned in Table IV.

The sort FOR is for TCRL terms that denote CAL formulas; M_R and M_C are sorts for terms that denote, respectively, a part of the system state and a complete system state; T is the sort for terms that denote time points; and, finally, Γ is the sort for terms that denote system traces. Note that the order of sorts preserves under the import relation between languages. For example, the following orders exist among the imported sorts in TCRL: $\mathrm{SP}_{\mathrm{comp}} < \mathrm{SP}$, $\mathrm{SP}_{\mathrm{interface}} < \mathrm{SP}_{\mathrm{comp}}$, $\mathrm{SP}_{\mathrm{input}} < \mathrm{SP}_{\mathrm{interface}}$, $\mathrm{SP}_{\mathrm{output}} < \mathrm{SP}_{\mathrm{interface}}$. Furthermore, DEFOR $<$ FOR and ΣFOR $<$ FOR.

For all sorts, a set of constants and a set of variables are assumed. For all n-ary functions $f \in$ func(CAL) and all sorts $S, S_1, \ldots, S_n \in$ sort(CAL), where f: $S_1 \times \cdots \times S_n \rightarrow S$, the function $\underline{f}$: $S_1 \times \cdots \times S_n \rightarrow S$ is an imported function in TCRL. Also, for all n-ary predicate $p \in$ pred(CAL) and all sorts $S_1, \ldots, S_n \in$ sort(CAL), where p: $S_1 \times \cdots \times S_n$, predicate p is reformulated as a function $\underline{p}$: $S_1 \times \cdots \times S_n \rightarrow$ FOR and imported in TCRL. Finally, logical connectives that operate on CAL formulas are imported as functions, e.g.,

$$\wedge\text{: FOR} \times \text{FOR} \rightarrow \text{FOR}$$

$$\neg\text{: FOR} \rightarrow \text{FOR}$$

Table V. Predicates of TCRL.

Predicate	Description
$<: T \times T$	Time-ordering relation
holds_info_r: $M_R \times$ FOR $\times$ SIGN	Formula holds a truth value indicated by sign in a state
holds_info_c: $M_C \times$ FOR $\times$ $\mathrm{SP}_{\mathrm{interface}} \times$ SIGN	Formula holds a truth value indicated by sign in an interface state of a component
holds_struct_r: $M_R \times$ DEFOR $\times$ SIGN	Design formula holds a truth value indicated by sign in a state
holds_struct_c: $M_C \times$ DEFOR $\times$ SP $\times$ SIGN	Design formula holds a truth value indicated by sign in a state of a component

Some specific functions for the trace requirement language TCRL are as follows:

$$\begin{aligned} &\textit{input:} && \mathrm{SP}_{\mathrm{comp}} \rightarrow \mathrm{SP}_{\mathrm{input}} \\ &\textit{output:} && \mathrm{SP}_{\mathrm{comp}} \rightarrow \mathrm{SP}_{\mathrm{output}} \\ &\textit{state\_r}: && \Gamma \times T \times \mathrm{SP}_{\mathrm{interface}} \rightarrow M_R \\ &\textit{state\_c}: && \Gamma \times T \rightarrow M_C \end{aligned}$$

Given the functions as defined here, the terms of TCRL are defined in the usual way. To specify the behavior of the system through time, predicates are introduced that express the validity of formulas regarding either the information state of the system (FOR) or the state of configuration of the system (DEFOR) at particular time points. These predicates are hold_info_r, hold_struct_r, hold_info-c, and hold_struct_c. The difference between these predicates is that the "_c" predicates express the validity of a formula with respect to a complete state whereas the "_r" predicates express the validity of a formula with respect to a partial state. This distinction is useful because one and the same formula from FOR can be used by different components and thus have different truth values in those different components. The "_info_" predicates refer to the validity of a formula with respect to information states; "_struct_" predicates refer to structural states of the system (system configuration). These predicates are mentioned in Table V.

Formulas of TCRL are defined in the usual way:

1. If p: $S_1 \times \cdots \times S_n$ is a n-ary predicate and t_i is a term of sort S_i ($i = 1, \ldots, n$), then $p(t_1, \ldots, t_n)$ is an expression of the requirement language TCRL.
2. If E_1 and E_2 are TCRL 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 formulas of TCRL.

Let δ be a design formula of sort DEFOR, $\mathrm{agent}_{\mathrm{MA}}$ is the MA, let SYS denote the whole multiagent system, and then the following TCRL formula states that in all system traces and at any time point if the output of the MA is a communication formula expressing a creation action (i.e., to_be_performed(add(δ))); then, some time later the system is modified according to the creation action:

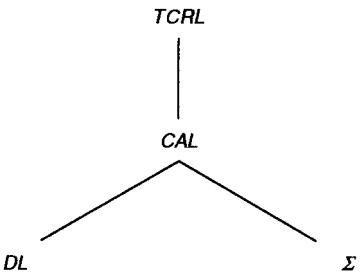

Figure 4. Import relation between languages.

$$\forall\, \gamma\colon \Gamma, \exists\, t\colon T\,[\text{holds\_info\_}c(\text{state\_}c(\gamma\colon \Gamma, t\colon T), \text{to\_be\_performed}(\text{add}(\delta)), \text{output}(\text{agent}_{\text{MA}}), \text{true}) \Rightarrow \exists\, t'\colon T\,[t'\colon T > t\colon T \wedge \text{holds\_struct\_}c(\text{state\_}c(\gamma\colon \Gamma, t'\colon T), \delta, \text{SYS}, \text{true}))]$$

In the following, quantified expressions such as "$\exists\, t'\colon T, t'\colon T > t\colon T \wedge \cdots$" are abbreviated to "$\exists\, t'\colon T > t\colon T \cdots$".

Note that there is an ordering relation between defined languages (DL, CAL, Σ, and TCRL) according to which one language imports terms and formulas from another language. This import ordering relation is illustrated in Figure 4.

6. REQUIREMENTS FOR AN EXAMPLE SCENARIO

In this section, we use the requirement language TCRL to express a number of requirements that specify relevant properties concerning both configuration as well as information behavior of agent-mediated systems. To motivate these requirements, we introduce a scenario where the system configuration is modified.

6.1. Example Scenario

Consider a scenario for a system consisting of three components called User Agent (UA), MA, and EW. At a certain time point, the UA may need a PA agent and communicates an appropriate request to the MA. The MA generates an action (to be executed in the environment) to create a PA agent for the UA. After the PA agent is created (now the system consists of four components), the UA can communicate with its PA agent and require certain information to be provided by the PA 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 PA.
2. System structure consists of UA, MA, and EW; UA generates a service request on its output for PA.

Table VI. State transition operations.

Operation	In scenario	Explication
Communication initiation	1 → 2	UA: need for PA
	6 → 7	UA: specific info need
	8 → 9	PA: fitting answer
Communication event	2 → 3	UA to MA: requests a PA
	7 → 8	UA to PA: specific info request
	9 → 10	PA to UA: fitting answer
Action initiation	3 → 4	MA decides to create PA
(World) interaction event	4 → 5	MA to EW: add(PA)
Action execution	5 → 6	EW executes add(PA)

3. System structure consists of UA, MA, and EW; MA has the user service request for PA 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; UA (internally) identifies which specific information is needed.
7. System structure consists of UA, MA, EW, and PA; UA generates information request on its output.
8. System structure consists of UA, MA, EW, and PA; PA has the information request on its input; PA (internally) identifies a fitting answer.
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 foregoing sequence of system states, the following types of state transition operations can be distinguished: communication initiation, communication event, action initiation, (world) interaction event, and action execution. In Table VI the state transition operations within the example scenario are explicated.

6.2. Relevant Properties

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

6.2.1. Global Properties

Two important global properties for the example mentioned previously are specified by 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 such an E on its output.

$\forall\ \gamma$: Γ, t: T, Q: SLTERM
[holds_info_r(state_r(γ: Γ, t: T, output(UA)),
communication_from_to(Q: SLTERM, UA, MA), true)
$\wedge\ \exists\ E$: DEFOR structure_realizes_service(E: DEFOR, Q: SLTERM)
$\Rightarrow$
$\exists\ t'$: $T > t$: $T\ \exists\ E$: DEFOR
structure_realizes_service(E: DEFOR,
Q: SLTERM)
$\wedge$ holds_info_r(state_r(γ: Γ, t': T, output(EW)), E: DEFOR, true)
]

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 a PA) to MA on its output, and R is the behavior 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 behavior R.

$\forall\ \gamma$: Γ, t: T, Q: SLTERM
[holds_info_r(state_r(γ: Γ, t: T, output(UA)),
communication_from_to(Q: SLTERM, UA, MA), true)
$\wedge\ \exists\ E$: DEFOR, $d1$: T, $d2$: T
[structure_realizes_service(E: DEFOR, Q: SLTERM) $\wedge$
$SB1_R$(E: DEFOR, $d1$: T, $d2$: T)]
$\Rightarrow$
$\exists\ t'$: $T > t$: T, E': DEFOR, $d1$: T, $d2$: T
[structure_realizes_service(E': DEFOR, Q: SLTERM)
$\wedge$ holds_info_r(state_r(γ: Γ, t': T, output(EW)), E': DEFOR, true)
$\wedge$ $SB1_R$(E': DEFOR, $d1$: T, $d2$: T)
$\wedge\ \forall\ t1$: T, $t2$: T, $t3$: T, $t4$: T
[$t2$: $T - t1$: $T \geqslant d1$: T
$\wedge\ t1$: $T \leqslant t3$: $T \leqslant t2$: $T - d2$: T
$\wedge\ t4$: $T - t3$: $T \geqslant d2$: T
$\Rightarrow R$(γ: Γ, $t3$: T, $t4$: T, pers_ass)]]
]

The term $SB1_R$ is a scheme of properties that relate a structure E: DEFOR to behavior R. The occurrence of R makes it a scheme. The terms $SB1_R$ and R occurring in this property are defined and explained in Section 6.2.9.

6.2.2. Basic Properties

The basic properties are divided in three subclasses called agent properties, world properties, and transfer properties. These properties are defined as follows.

6.2.3. Agent Properties

An agent property refers to the behavior of a specific agent. An important agent property for the example mentioned previously 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 PA) on its input, and there exists an E which is a system configuration that can satisfy the service request Q, then a time point $t' > t$ exists such that MA for such an E has to_be_performed(add(E)) on its output.

$\forall\ \gamma$: Γ, t: T, Q: SLTERM,
[holds_info_r(state_r(γ: Γ, t: T, input(MA)),
communication_from_to(Q: SLTERM, UA, MA), true)
$\wedge\ \exists$ E: DEFOR structure_realizes_service(E: DEFOR, Q: SLTERM)
$\wedge\ \neg\ \exists\ E'$: DEFOR
[holds_info_r(state_r(γ: Γ, t: T, input(MA)),
observation_result(E': DEFOR, pos), true)
$\wedge$ structure_realizes_service(E': DEFOR, Q: SLTERM)]
$\Rightarrow$
$\exists\ t'$: $T > t$: $T\ \exists$ E: DEFOR
[structure_realizes_service (E: DEFOR, Q: SLTERM)
$\wedge$ holds_info_r(state_r(γ: Γ, t': T, output(MA)),
to_be_performed(add(E: DEFOR)), true)
$\wedge\ \forall\ E''$: DEFOR
[structure_realizes_service(E'': DEFOR, Q: SLTERM)
$\wedge$ holds_info_r(state_r(γ: Γ, t': T, output(MA)),
to_be_performed(add(E'': DEFOR)), true)]
$\wedge$ equal(E: DEFOR, E'': DEFOR)]
]

6.2.4. World Properties

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

ER1 (EW Action Execution Successfulness). If at time point t the world component EW has to_be_performed(α) on its input, and β is the effect of performing α, then a time point $t' > t$ exists such that EW has β on its output.

$\forall\ \gamma$: Γ, t: T, α: ACTION, β: DEFOR,
[holds_info_r(state_r(γ: Γ, t: T, input(EW)),
to_be_performed(α: ACTION), true)
$\Rightarrow$
$\exists\ t'$: $T > t$: T
[holds_info_r(state_r(γ: Γ, t': T, output(EW)), β: DEFOR, true)
$\wedge$ is_effect_of(β: DEFOR, α: ACTION)]
]

where the predicate is_effect_of(β, α) is defined as follows:

$\forall\ x$: DEFOR$^+$ is_effect_of(x, add(x))
$\forall\ x$: DEAT is_effect_of($\neg x$, delete(x))

6.2.5. *Transfer Properties*

The transfer properties specify that information transfer between agents takes place in a proper manner. Two important transfer properties are specified by 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

$\forall\ \gamma$: Γ, t: T, φ: FOR
[holds_info_r(state_r(γ: Γ, t: T, output(UA)),
communication_from_to(φ: FOR, UA, MA), true)
$\Rightarrow$
$\exists\ t'$: $T > t$: T
holds_info_r(state_r(γ: Γ, t': T, input(MA)),
communication_from_to(φ: FOR, UA, MA), true)
]

Note that the property TR1 also could be specified using a variables over SP instead of the specific references to output(UA) and input(MA). The TR1 property then can be instantiated for every information transfer between components by instantiating the proper interfaces of those components for the variables over SP.

TR2 (Transfer MA-EW). If at time point t the agent MA generates to_be_performed(α) for EW on its output (Note that α is of type ACTION such that it is either add(E) or delete(E)), then a time point $t' > t$ exists such that EW has to_be_performed(α) on its input.

$\forall\ \gamma$: Γ, t: T, α: ACTION
[holds_info_r(state_r(γ: Γ, t: T, output(MA)),
to_be_performed(α: ACTION), true)
$\Rightarrow$
$\exists\ t'$: $T > t$: T

holds_info_r(state_r(γ: Γ, t': T, input(EW)),
to_be_performed(α: ACTION), true)
]

6.2.6. *Semantic and Coherence Properties*

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

6.2.7. *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 γ the world component EW has description E on its output, then at time point t in trace γ the system has structure E (i.e., the system description E is true).

$\forall$ γ: Γ, t: T, E: DEFOR
[holds_info_r(state_r(γ: Γ, t: T, output(EW)), E: DEFOR, true)
$\Rightarrow$
holds_struct_c(state_c(γ: Γ, t: T), E: DEFOR, true)]

6.2.8. *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 γ 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).

$\forall$ γ: Γ, t: T, C: SP, φ: FOR, E: DEFOR, tv: TV
[holds_info_r(state_r(γ: Γ, t: T, interface(C: SP)), φ: FOR, tv: TV)
$\Rightarrow$
$\exists$ Σ: SSIG
[holds_struct_r(state_r(γ: Γ, t: T, SYS), exists_comp(C: SP), true)
$\wedge$ holds_struct_r(state_r(γ: Γ, t: T, SYS),
has_interface_signature(Σ: SSIG, C: SP), true)
$\wedge$ holds_struct_r(state_r(γ: Γ, t: T, SYS),
is_formula_of(φ: FOR, Σ: SSIG), true)]
]

CA2. At any time t in any trace γ if the system has a certain structure (in its structural state), then the formulas that can be evaluated in this system structure have truth values (in the system's informational state).

$\forall\ \gamma$: Γ, t: T, C: SP, φ: FOR, E: DEFOR, Σ: SSIG
[[holds_struct_r(state_r(γ: Γ, t: T, SYS), exists_comp(C: SP), true)
$\wedge$ holds_struct_r(state_r(γ: Γ, t: T, SYS),
has_interface_signature(Σ: SSIG, C: SP), true)
$\wedge$ holds_struct_r(state_r(γ: Γ, t: T, SYS),
is_formula_of(φ: FOR, Σ: SSIG), true)]
$\Rightarrow$
$\exists\ tv$: TV
holds_info_r(state_r(γ: Γ, t: T, interface(C: SP)), φ: FOR, tv: TV)
]

6.2.9. *Structure-Behavior Properties*

In this section (required), behavior is related to structure properties and service requests. In this section $R(\gamma$: Γ, $t1$: T, $t2$: T, C: SP) stands for a scheme of behavioral requirements that can be instantiated by a specific requirement. As an example, pers_ass_req(γ: Γ, $t1$: T, $t2$: T, C: SP) denotes the requirement that if a component C receives a request on its input, then the component C produces an answer within a reasonable time for that request on its output. In the formalization of this requirement INFO is a sort for the content of requests and answers:

$\forall\ A$: INFO
[holds_info_r(state_r(γ: Γ, $t1$: T, input(C: SP)), request(A: INFO), true)
$\Rightarrow$
$\exists\ t$: T, $\exists\ B$: INFO
[$t1$: $T \leqslant t$: $T \leqslant t1$: $T + t2$: T
$\wedge$ holds_info_r(state_r(γ: Γ, t: T, output(C: SP)),
answer_for(B: INFO), true)]
]

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

Let $SB1_R$ (E: DEFOR, $d1$: T, $d2$: T) denote the following scheme of structure-behavior 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):

$\forall\ \gamma$: Γ, $\forall\ t1$: T, $t2$: T, $t3$: T, $t4$: T
[$t2$: $T - t1$: $T \geqslant d1$: T
$\wedge\ \forall\ t \in [t1$: T, $t2$: $T]$
holds_struct_c(state_c(γ: Γ, t: T), E: DEFOR, true)
$\wedge\ t1$: $T \leqslant t3$: $T \leqslant t2$: $T - d2$: T
$\wedge\ t4$: $T - t3$: $T \geqslant d2$: T
]

$\Rightarrow$
$R(\gamma\colon \Gamma,\ t3\colon T,\ t4\colon T,\ \text{pers\_ass})$
]

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

$\forall\ Q$: SLTERM
[intended_service$_R(Q$: SLTERM)
$\Rightarrow$
$\exists\ E$: DEFOR, $d1$: T, $d2$: T
[$SB1_R(E$: DEFOR, $d1$: T, $d2$: $T)$
$\wedge$ structure_realizes_service(E: DEFOR, Q: SLTERM)]
]

This scheme expresses that for all service requests Q, if by expressing Q the user intends to obtain the service specified by R, then a system configuration E exists such that the structure E can be used to realize the requested service Q, and that ($SB1_R$) the intended service R will indeed be delivered under the condition that the system satisfies configuration E during an appripriate time.

7. SYSTEM EVALUATION

Development of a system is only then complete when the system behavior is verified against the required properties. In Section 3, certain properties for the proposed scenario system have been introduced. Verifying such properties may be quite complex. This section shows that such complexity can be reduced by introducing appropriate intermediate properties. By means of the intermediate properties the global properties can be proved more easily and more efficiently by using proof patterns. The intermediate properties can be derived from basic properties.

7.1. Intermediate Properties

In this article, intermediate properties are chosen so that they specify the output/output relations between different system components. The reason for formulating them in that format is the ease with which they can be used in proof patterns. 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, and there exists an E which is a structure with which Q can be realized, then, if necessary to service the request, a time point $t' > t$ exists such that MA for such an E has to_be_performed(add(E)) on its output.

$\forall\ \gamma\colon \Gamma,\ t\colon T,\ Q\colon$ SLTERM

[holds_info_r(state_r(γ: Γ, t: T, output(UA)),
communication_from_to(Q: SLTERM, UA, MA), true)
$\wedge$ $\exists$ E: DEFOR structure_realizes_service(E: DEFOR, Q: SLTERM)
$\wedge$ $\neg$ $\exists$ E': DEFOR
[holds_info_r(state_r(γ: Γ, t: T, output(EW)),
observation_result(E': DEFOR, pos), true)
$\wedge$ structure_realizes_service(E': DEFOR, Q: SLTERM)]
$\Rightarrow$
$\exists$ t': $T > t$: T, E: DEFOR structure_realizes_service
(E: DEFOR, Q: SLTERM) $\wedge$
[holds_info_r(state_r(γ: Γ, t': T, output(MA)),
to_be_performed(add(E: DEFOR)), true)
$\wedge$ $\forall$ E'': DEFOR
[structure_realizes_service(E'': DEFOR, Q: SLTERM)
$\wedge$ holds_info_r(state_r(γ: Γ, t': T, output(MA)),
to_be_performed(add(E'': DEFOR)), true)
$\Rightarrow$
equal(E: DEFOR, E'': DEFOR)
]
]
]

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 the output of EW reflects the effect of the action (e.g., E is, respectively, is not on its output).

$\forall$ γ: Γ, t: T, α: ACTION, β: DEFOR
[holds_info_r(state_r(γ: Γ, t: T, output(MA)),
to_be_performed(α: ACTION), true)
$\wedge$ is_effect_of(β: DEFOR, α: ACTION)
$\Rightarrow$
$\exists$ t': $T > t$: T
holds_info_r(state_r(γ: Γ, t': T, output(EW)), β: DEFOR, true)
]

7.2. 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:
   TR1 & AR1 $\Rightarrow$ IR1
   TR2 & ER1 $\Rightarrow$ IR2
2. Intermediate properties can be chained to indirect interaction properties expressing indirect impact of one component on another one:
   IR1 & IR2 $\Rightarrow$ GR1

3. Indirect intermediate properties imply the global properties, assuming representation and coherence properties and structure-behavior relationships:

$$GR1 \,\&\, RA1 \,\&\, CA1 \,\&\, SB2 \Rightarrow GR2$$

7.3. Checking Properties

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

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

For more details of this model-checking environment and its use in diagnosis, see Ref. 12.

8. DISCUSSION

The requirements specification language introduced in this study 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., Refs. 7, 13 and 14) only allow for specification of informational system states, for a given, fixed system structure. In our language it is possible to refer to both the structural state of the system and the informational state of it.

A second use is that a broad class of behavioral properties can be specified of the system or of specific parts of the system, e.g., agents. Not only can, e.g., reactiveness and proactiveness properties be specified, but also properties expressing adaptive behavior, 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 can not 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, e.g., as initiated and performed by the system (e.g., one of its agents) itself. Here, requirements on, e.g., agent behavior and the dynamics of the system structure and their relationships 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, logically imply the global requirement. For more details of refinement in the context of compositional verification of multiagent systems, see Ref. 10.

For all different types of requirements discussed and 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 a 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) (see Ref. 12).

References

1. Davis AM. Software requirements: Objects, functions, and states. Englewood Cliffs, NJ: Prentice Hall; 1993.
2. Kontonya G, Sommerville I. Requirements engineering: Processes and techniques. New York: John Wiley and Sons; 1998.
3. Sommerville I, Sawyer P. Requirements engineering: A good practice guide. Chicester, England: John Wiley & Sons; 1997.
4. Dardenne A, Lamsweerde A van, Fickas S. Goal-directed requirements acquisition. Sci Comput Program 1993;20:3–50.
5. Dubois E, Du Bois P, Zeippen JM. A formal requirements engineering method for real-time, concurrent, and distributed systems. In: Proc of the Real-Time Systems Conf, RTS'95, 1995.
6. Engelfriet J, Jonker CM, Treur J. Compositional verification of multi-agent systems in temporal multi-epistemic logic. J Logic, Lang Inf 2002;11:195–225.
7. Fisher M, Wooldridge M. On the formal specification and verification of multi-agent systems. Int J Coop Inf Syst, M. Huhns, M. Singh, editors. Special Issue on Formal Methods in Cooperative Information Systems: Multi-Agent Systems, 1997;6:67–94.
8. Manna Z, Pnueli A. Temporal verification of reactive systems: Safety. Berlin: Springer-Verlag; 1995.
9. Brazier FMT, Jonker CM, Treur J. Dynamics and control in Component-Based Agent Models. Int J Intell Syst., 2002;17:1007–1068.
10. Jonker CM, Treur J. Compositional verification of multi-agent systems: A formal analysis of pro-activeness and reactiveness. Int J Coop Inf Syst 2002;11:51–92.
11. Brazier FMT, Jonker CM, Treur J. Principles of component-based design of intelligent agents. Data Knowl Eng 2002;41:1–28.
12. Jonker CM, Letia IA, Treur J. Diagnosis of the dynamics within an organization by trace checking of behavioral requirements. In: Wooldridge M, Weiss G, Ciancarini P, editors. Proc of the 2nd Int Workshop on Agent-Oriented Software Engineering, AOSE'01. Lecture Notes in Computer Science, Vol. 2222. Berlin: Springer Verlag; 2002. pp 17–32.
13. Darimont R, Lamsweerde A van. Formal refinement patterns for goal-driven requirements elaboration. In: Proc 4th ACM Symposium on the Foundation of Software Engineering (FSE4), 1996. pp 179–190.
14. Herlea DE, Jonker CM, Treur J, Wijngaards NJE. Specification of behavioural requirements with compositional multi-agent system design. In: Garijo FJ, Boman M, editors. 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, Berlin: Springer Verlag, 1999. pp 8–27.