REQUIREMENTS SPECIFICATION AND AUTOMATED EVALUATION OF DYNAMIC PROPERTIES OF A COMPONENT-BASED DESIGN

C.M. JONKER, J. TREUR, W.C.A. WIJNGAARDS
Vrije Universiteit Amsterdam, The Netherlands

Abstract. Within a design process, the evaluation of a candidate design solution against a set of requirements may be hard, especially when the requirements concern dynamic properties. For a component-based design, evaluation of the dynamics can be based on dynamic properties of the components, and the way in which they are connected. In this paper an automated approach to the evaluation of dynamic properties of a component-based design is presented. A declarative temporal modelling language to specify and analyse dynamic properties is offered. An executable subset of this language is defined, based on 'leads to' relations. If executable specifications in terms of leads to relations of dynamic properties of the (reusable) components within a component-based design are available, then automated support is offered for: (1) simulation of the overall dynamics based on the executable dynamic properties of the components, (2) evaluation of requirements in the form of required overall dynamic properties of a design against a number of execution traces of the design, and (3) proving properties of an overall component-based design from executable properties of its components. The paper presents methods, techniques and software for such support and illustrates these by an example from the application area of component-based software design.

1. Introduction

In some application areas of design, systems which have nontrivial dynamics are designed in a component-based manner. An example of such an application area is the design of component-based software for dynamic applications. In such application areas often components can be (re)used for which the dynamic properties are known. By composing a number of such components in a component-based design, the required overall dynamics is obtained. If the dynamics required is not that simple, it is not straightforward how such dynamics relates to available reusable components and their

dynamic properties. Therefore automated support for such design processes is desirable. Literature addressing automated support in the areas of model checking and verification in general can be found, for example, in (Henzinger, Nicollin, Sifakis and Yovine 1994; Bouajjani, Lakhnech, Yovine 1996; Yovine 1997; Fisher 1994; Clarke, Grumberg and Peled 2000; Manna and Pnueli 1995; Stirling 2001).

To support component-based design of dynamical systems, combining required nontrivial dynamics with the use of reusable components, a number of questions need to be addressed:

- How can dynamic properties of (reusable) components be specified?
- How can requirements on the dynamics of an overall design be specified?
- How can it be checked whether a given component-based design, with known dynamic properties of its components, fulfils a given requirement on its overall dynamics?
- To fulfill overall requirements on dynamics, how can proper reusable components be determined on the basis of their dynamic properties, and how can they be composed to obtain a component-based design fulfilling these overal requirements?

In this paper a rich temporal framework is used to address these questions (with an emphasis on the first three). The temporal framework provides a declarative (requirement) specification environment and is used for a variety of activities, such as temporal simulation, temporal model checking and temporal reasoning about dynamic properties. The application area used to illustrate the approach is the design of software, in particular a system of information agents.

Specification of dynamic properties of a component-based design has at least two different aspects of use. First, models for the dynamics can be specified to be used as a basis for *simulation*, also called executable models. These types of models can be used to perform (pseudo-)experiments on the basis of the design. Second, specification of dynamic properties of a system can be done in order to *analyse* its dynamics (i.e., to obtain automated support for requirements specification, verification and testing). These properties can play the role of requirements, and can be used, for example, in evaluation of sample behaviours of (realised or simulated) designs in the context of these requirements. These two different uses of specification of dynamic properties of a component-based design impose different desiderata on the languages in which these specifications are to be expressed.

A language for executable models should be formal, and as simple as possible, to avoid computational complexity. Expressivity can be limited. Software tools to support such a language serve as *simulation environment*. A language to analyse dynamic properties, on the other hand, should be sufficiently advanced to express various dynamic properties that are

relevant. Expressivity should not be too limited; executability, however, is not required for such a language. What is important, though, is that properties specified in such a language can be checked for a given sample behaviour (e.g., a simulation run) without much work, preferably in an automated manner. Moreover, it is useful if a language to specify dynamic properties provides possibilities for further analysis of logical relationships between properties. For these reasons also a language to specify properties of the dynamics of a design should be formal, and at least partly supported by software tools (*analysis environment*).

In this paper two different temporal specification languages for dynamic properties are put forward and illustrated for an example application. In Sections 2 and 3 it is shown how the two languages (one aiming at analysis, the other one aiming at simulation) to model dynamics within a component-based design can be defined. An example component-based software design is used to illustrate the notions that are introduced. In Section 4 a model for the dynamics of this example design is presented. Section 5 addresses the use of the analysis environment for the example design. In Section 6 the use of the simulation environment is addressed. In Section 7 a discussion is included.

2. Specification of Dynamic Properties of a Component-Based Design

To specify dynamic properties of a design, the temporal trace language used in (Jonker and Treur 1998; Herlea, Jonker, Treur and Wijngaards 1999) is adopted. This is a language in the family of languages to which also situation calculus (McCarthy and Hayes 1969; Reiter, 2001), event calculus (Kowalski and Sergot 1986), and fluent calculus (Hölldobler and Thielscher 1990) belong. In Section 2.1 the Temporal Trace Language TTL is introduced. This language is more expressive than modal and temporal languages as described, for example, in (Henzinger, Nicollin, Sifakis and Yovine 1994; Bouajjani, Lakhnech and Yovine 1996; Yovine 1997; Fisher 1994; Clarke, Grumberg and Peled 2000; Manna and Pnueli 1995; Stirling 2001); see Section 6 for more discussion about expressivity. In Section 2.2 an example component-based software design (of a multi-agent system of information agents) is introduced to illustrate the language and its use.

2.1. THE TEMPORAL TRACE LANGUAGE TTL

An *ontology* is a specification (in order-sorted logic) of a vocabulary (also called a signature). A state for ontology Ont is an assignment of truth-values {true, false} to the set of ground atoms At(Ont). The *set of all possible states* for ontology Ont is denoted by STATES(Ont). The standard satisfaction relation |= between states and state properties is used: S |= p means that state property p holds in state S.

To describe behaviour, explicit reference is made to time in a formal manner. A fixed *time frame* T is assumed which is linearly ordered. Depending on the application, it may be dense (e.g., the real numbers), or discrete (e.g., the set of integers or natural numbers or a finite initial segment of the natural numbers), or any other form, as long as it has a linear ordering. A *trace* $\mathcal{T}$ over an ontology Ont and time frame T is a mapping $\mathcal{T}$: T → STATES(Ont), i.e., a sequence of states $\mathcal{T}_t$ (t ∈ T) in STATES(Ont). The set of all traces over ontology Ont is denoted by TRACES(Ont), i.e., TRACES(Ont) = STATES(Ont)^T.

States of a trace can be related to state properties via the formally defined satisfaction relation |= between states and formulae. Comparable to the approach in situation calculus, the sorted predicate logic *Temporal Trace Language* TTL is built on atoms referring to traces, time and state properties, such as

state($\mathcal{T}$, t, output(C)) |= p.

This expression denotes that state property p is true at the output of component C in the state of trace $\mathcal{T}$ at time point t. Here |= is a predicate symbol in the language (in infix notation), comparable to the Holds-predicate in situation calculus. Temporal formulae are built using the usual logical connectives and quantification (for example, over traces, time and state properties). The set TFOR(Ont) is the set of all *temporal formulae* that only make use of ontology Ont. We allow additional language elements as abbreviations of formulae of the temporal trace language. Ontologies can be specific for a component.

2.2 AN EXAMPLE COMPONENT-BASED SOFTWARE DESIGN

For the example, consider a component-based design for a system of two software agents A and B (represented in the design by two components; see Figure 1) that participate in a small project: they each have to acquire some information (in the External World, represented by a third component in the design) and in cooperation they make up a concluding report on some topic. Each of the agents has access to useful sources of information, but this differs for the two agents. By co-operation they can benefit from the exchange of information that is only accessible to the other agent. If both types of information are combined, the relevant conclusions can be drawn that would not have been achievable for each of the agents separately. A relevant property of such a component-based design is successfulness:

- What makes the difference for such a co-operation to succeed or to fail?

- Which dynamic properties of the components are relevant for being successful?
- How does successfulness relate to these dynamic properties?

For example, one of the agents, say A, may not be pro-active in its individual search for information. This might be compensated if the agent B is pro-active in requesting the other agent for information, but then at least A has to be reactive (and not entirely inactive) in information acquisition. Also other reasons for failure may exist. For example, one of the agents may not be willing to share its acquired information with the other agent. Yet another reason for failure may be that although both agents are active in searching and exchanging information, none of them is combining different types of information and deduce new conclusions. So, dynamic properties of the design as a whole depend in a crucial manner on the dynamic properties of the components, in this case the proper pro-activeness and reactiveness properties that play a role in the dynamics of the cooperation process of the agents.

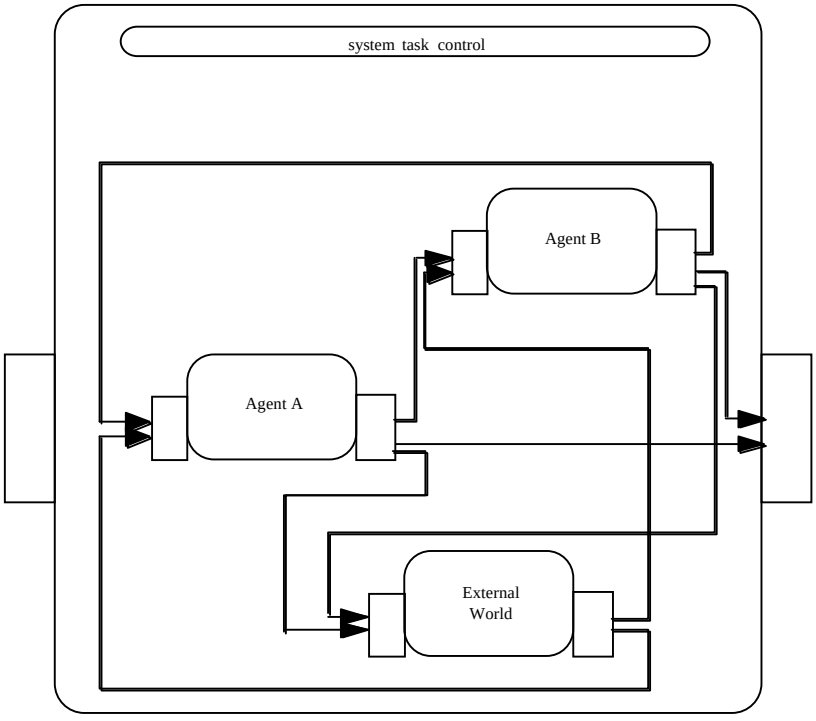

Figure 1. The Example Component-Based Design

Summarizing, this example component-based design as depicted in Figure 1 is composed of three components: two information agents A and B and a component EW representing the External World. In this figure the boxes denote system components. During processing of the system, each component has at each point in time an input state and an output state. The arrows depict output-input connections: channels taking care that information available at an output state is also made available (possibly with some small delay) for the connected input state. Each of the agents is able to acquire partial information in the External World by initiated information acquisition, this is modelled by an arrow from the agent to the External World transferring information on what is to be acquired, and by an arrow back transferring information on the results of the acquisition.

Each agent's own observations are insufficient to draw conclusions of a desired type, but the combined information of both agents is sufficient: they have to co-operate to be able to draw conclusions. Therefore communication is required; each agent can communicate results of its own information acquisition and requests for information to the other agent: the arrows between the agents.

Assumptions on the notion of component-based design

The above description gives an indication of the – very general – assumptions that are made on the notion of component-based design. This is a design structure based on *components* with at each point in time *input states* and *output states* that are *connected* by arrows to form the design. An arrow *modifies* the destination state at each point in time that the source state has changed, by inserting the changed source state in it.

For reasons of presentation, this by itself quite common situation for co-operative information agents is materialised in the following more concrete form. The world situation consists of an object that has to be classified. One agent can observe only the bottom view of the object, the other agent the side view. By exchanging and combining acquired information on the object they are able to classify the object. For example, if A observes a circle, and B observes a square, then, if A and B cooperate, together they can conclude that the object is a cylinder.

Communication from the agent A to B takes place in the following manner:

- the agent A generates at its output state a statement of the form:

  communication_from_to(<*type*>, <*atom*>, <*sign*>, A, B)
- this information is transferred to the input state of B using the arrow from A to B

In the example <*type*> can be filled with a label request or world_info, <*atom*> is an atom expressing information on the world, and <*sign*>, is one of pos or neg, to indicate truth or falsity. Instances of communication information that can be expressed are:

communication_from_to(request, view(B,circle), pos, A, B)
communication_from_to(world_info, view(B,circle), neg, B, A)

Here the object atom view(B,circle) expresses the world information that the view of the object visible for B is a circle. Interaction between an agent A and the External World takes place as follows:

- the agent A generates at its output state a statement of the form:
  to_be_acquired_by(<*atom*>, A)
- the information is transferred to the input state of EW
- the External World EW generates at its output state a statement of the form:
  acquired_information_for(<*atom*>, <*sign*>, A)
- the information is transferred to A

Instances of information on information acquisition for an agent A that can be expressed are:

to_be_acquired_by(view(A,circle), A)
acquired_information_for(view(A,circle), pos, A)

The output of an agent can include conclusions about the classification of the object of the form conclusion(object_type(O)); these are transferred to the output of the system as a whole.

In Table 1 an informal sketch of an example trace showing the dynamics of the component-based design is depicted. Here for simplicity the transfer of information between components is assumed to take no time (in contrast, in the simulation presented in Section 6, the transfer does take time). In this example trace, at time point 1 agent A takes two initiatives (see A's output state): (1) to acquire information and (2) to request information from B. These initiatives are immediately (same time point) received at the input of the External World component and at the input of B. As a result the External World provides the information (output at time point 2), and agent B starts

an information acquisition process (output at time point 2). Immediately (at time point 2) the acquired information is received by A's input, and the information acquisition initiation of B is received at the External World component's input. As a result, the External World provides the information for B (time point 3), and this immediately is transferred to B's input. As a result, B communicates this information to A (time point 4). Finally, A, combining the gathered information, draws the conclusion.

TABLE 1. An example trace of a cooperative information gathering process

Time point	1	2	3	4	5
Agent A input		information acquired by A		information acquired by B for A	
output	acquisition initiation by A; request of A to B				conclusion combining the two types of information
Agent B input	request of A to B		information acquired by B		
output		acquisition initiation by B		information acquired by B for A	
External World input	acquisition initiation by A	acquisition initiation by B			
output		information acquired by A	information acquired by B		

The example trace discussed above shows a successful cooperation. In virtue of which dynamic properties of the components was this success possible? This question will be addressed in Section 3. In the remainder of this section, to illustrate the use of the temporal trace language, some dynamic properties for this component-based design as a whole are shown. Expressed both informally and formally, two of these example properties state the following:

- *Successfulness*
  For any trace of the system, there exists a point in time such that in this trace at that point in time the system will provide a conclusion. Using the Temporal Trace Language TTL, this is formally expressed by:

$\forall \mathcal{T}$: TRACES $\exists$ t $\exists$ O : OBJECT
state($\mathcal{T}$, t, output(S)) $\models$ conclusion(object_type(O))

- *Cooperation necessity*
  For any trace of the system and any point in time, if in this trace agent A provides the relevant conclusion, then at an earlier point in time agent B has communicated to agent A information acquired by B.
  This is formally expressed by:
  $\forall \mathcal{T}$: TRACES $\forall$ O1 : OBJECT $\forall$ t
  state($\mathcal{T}$, t, output(A)) $\models$ conclusion(A, object_type(O1)) $\Rightarrow$
  $\exists$t' < t $\exists$ O2 : OUTLINE
  state($\mathcal{T}$, t, output(B)) $\models$ communication_from_to(world_info, view(B, O2), pos, B, A)

Other properties considered for this example are correctness and conservatism; see Section 5 for more details of these properties.

3. An Executable Language to Specify Dynamic Properties for Simulation

To obtain an executable language, in comparison with the temporal trace language discussed above strong constraints are imposed on what can be expressed. These constraints define a temporal language within the paradigm of executable temporal logic; cf. (Barringer et al. 1996). Roughly spoken, in this executable language it can only be expressed that if a certain state property holds for a certain time interval, then this *leads to* the situation that after some delay another state property should hold for a certain time interval. The use of real-valued parameters for time durations and delays is an extension, compared to the languages described in (Barringer et al. 1996). This specific temporal relationship $\bullet\!\!\rightarrow\!\!\!\!\rightarrow$ (*leads to*) is definable within the temporal trace language TTL. This definition is expressed in two parts, the forward in time part and the backward in time part. Time intervals are denoted by [x, y) (from and including x, to but not including y) and [x, y] (the same, but includes the y value).

Definition (The leads to relationship $\bullet\!\!\rightarrow\!\!\!\!\rightarrow$)
Let α and β be state properties, and let P1 and P2 refer to parts of the design (e.g., input or output of particular components). Then β *follows* α, denoted by $P1{:}\alpha \rightarrow\!\!\!\!\rightarrow_{e, f, g, h} P2{:}\beta$, with time delay interval [e, f] and duration parameters g and h if (see also Figure 2):

$\forall \mathcal{T}$: TRACES $\forall$t1:
[$\forall t \in [t1 - g, t1)$: state($\mathcal{T}$, t, P1) $\models \alpha \Rightarrow$
$\exists \lambda \in [e, f]\ \forall t \in [t1 + \lambda, t1 + \lambda + h)$: state($\mathcal{T}$, t, P2) $\models \beta$]

Conversely, the state property β *originates from* state property α, denoted by $P1{:}\alpha \bullet\!\!-_{e,f,g,h} P2{:}\beta$, with time delay in [e, f] and duration parameters g and h if

$\forall \mathcal{T}$: TRACES $\forall$ t2:

$[\forall t \in [t2, t2 + h) : state(\mathcal{T}, t, P2) \models \beta \Rightarrow$

$\exists \lambda \in [e, f] \forall t \in [t2 - \lambda - g, t2 - \lambda) \ state(\mathcal{T}, t, P1) \models \alpha]$

If both $P1{:}\alpha \twoheadrightarrow_{e,f,g,h} P2{:}\beta$, and $P1{:}\alpha \bullet\!\!-_{e,f,g,h} P2{:}\beta$ hold, this is called a *leads to* relation and denoted by $P1{:}\alpha \bullet\!\!\twoheadrightarrow_{e,f,g,h} P2{:}\beta$. Sometimes also conjunctions or negations on one of the sides (or both) of the arrow are used.

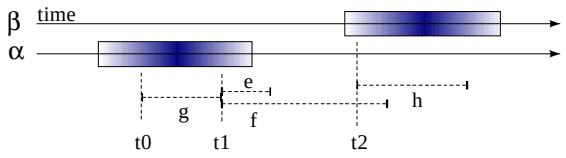

Figure 2. The Time Relationships between the Parameters

Notice that a special case is when the state property α is taken the trivial property true that is always (at every time point in every trace) true. In this case a leads to relation specifies that the property β will be *initiated* or *realized*. Examples of leads to relations in the context of the example component-based design as described in Section 2.2 are the following. Here the dynamic properties are formulated for a given agent A with respect to the other agent B; similar properties will hold for agent A's cooperation partner B.

Information acquisition reactive

A request to an agent A by another agent B for information that can be acquired by A leads to acquisition of this information by agent A.
Formally expressed:

$\forall$ O : OUTLINE

input(A):communication_from_to(request, view(A, O), pos, B, A) $\bullet\!\!\twoheadrightarrow_{5,5,10,10}$
output(A):to_be_acquired(view(A, O), A)

Request pro-active

Agent A initiates communication to agent B of a request for information that B is able to acquire.
Formally expressed:

$\forall$ O : OUTLINE

true $\bullet\!\!\twoheadrightarrow_{5,5,10,10}$ output(A) :communication_from_to(request, view(B, O), pos, A, B)

More examples can be found in Section 4.

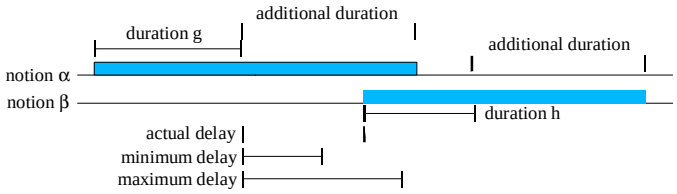

Figure 3. Temporal relationships for longer durations

The definition of the relationships as given above can be applied to situations where the sources hold for longer than the minimum interval length g. The result for a longer duration of α for P1:α $\bullet\!\!\rightarrow\!\!\!\!\rightarrow$ P2:β is depicted in Figure 3. The additional duration that the source holds, is also added to the duration that the result will hold, provided that the condition $e + h \geq f$ holds. This is because the definition can be applied at each subinterval of α, resulting in many overlapping intervals of β. The end result is that the additional duration also extends the duration that the resulting notion β holds.

4. An Executable Model for the Example Component-Based Design

To obtain an executable model for the dynamics of a component-based design, two types of dynamic properties in leads to format are used: properties for the dynamics of each of the components (Section 4.1), and properties that define the dynamics of the connections between components (Section 4.2).

4.1. DYNAMIC PROPERTIES OF THE COMPONENTS

For the example design, the components have dynamic properties as listed below. This does not mean that every possible component has all these properties. Rather, each specific candidate for a component is characterised by a subset of these properties. First the possible properties for the agents are discussed. They are expressed for agent A (with respect to the other agent B and the External World).

Information acquisition reactive
A request to an agent A by another agent B for information that can be acquired by A leads to acquisition of this information by agent A.
Formally expressed as:

$\forall$ O : OUTLINE

input(A) : communication_from_to(request, view(A, O), pos, B, A) $\bullet\!\!\rightarrow\!\!\!\!\rightarrow_{5,5,10,10}$

output(A) : to_be_acquired(view(A, O), A)

Information acquisition pro-active
Agent A initiates the acquisition of information it is able to acquire
Formally expressed as:

$\forall$ O : OUTLINE

true $\bullet\!\!\rightarrow\!\!\!\!\rightarrow_{5,5,10,10}$ output(A) : to_be_acquired(view(A, O), A)

Request pro-active
Agent A initiates communication to agent B of a request for information that B is able to acquire.
Formally expressed as:

$\forall$ O : OUTLINE

true $\bullet\!\!\rightarrow\!\!\!\!\rightarrow_{5,5,10,10}$ output(A) : communication_from_to(request, view(B, O), pos, A, B)

Information provision reactive
A request to an agent A by another agent B for information that can be acquired by A, together with receiving this acquired information by agent A leads to communication of this information to agent B.
Formally expressed as:

$\forall$ O : OUTLINE

input(A):communication_from_to(request, view(A, O), pos, B, A) $\wedge$
input(A) : acquired_information_for(view(A, O), pos, A) $\bullet\!\!\rightarrow\!\!\!\!\rightarrow_{5,5,10,10}$
output(A) : communication_from_to(world_info, view(A, O), pos, A, B)

Information provision proactive
Receiving acquired information by agent A leads to communication of this information to agent B.
Formally expressed:

$\forall$ O : OUTLINE

input(A) : acquired_information_for(view(A, O), pos, A) $\bullet\!\!\rightarrow\!\!\!\!\rightarrow_{5,5,10,10}$
output(A) : communication_from_to(world_info, view(A, O), pos, A, B)

Conclusion proactive
Receiving acquired information by agent A and receiving by communication information acquired by agent B leads to combining this information in drawing a conclusion. Which specific conclusion is to be drawn depends on the input information. Therefore, this property consists of a set of specific properties (instances), for example in formal form:

input(A) : acquired_information_for(view(A, square), pos, A) $\wedge$
input(A) : communication_from_to(world_info, view(B, square), pos, B, A) $\bullet\!\!\rightarrow\!\!\!\!\rightarrow_{5,5,10,10}$
output(A) : conclusion(object_type(cube))

The last property is a property of the External World component.

Information acquisition effectiveness
This is a property of the External World component that guarantees that every information acquisition initiative results in the acquired information Formally expressed:

$\forall$ X : AGENT $\forall$ O : OUTLINE $\forall$ S : SIGN
input(EW) : to_be_acquired(view(X, O), X) $\wedge$
output(EW) : true_in_world(view(X, O), S) $\bullet\!\!\rightarrow\!\!\!\!\rightarrow_{5,5,10,10}$
output(EW) : acquired_information_for(view(X, O), S, X)

4.2. DYNAMIC PROPERTIES FOR CONNECTIONS BETWEEN COMPONENTS

In addition to dynamic properties of components, connection properties are used to express the dynamics of the connections between the components in the design. These have a common pattern, instantiated for the particular components and the state properties involved at the connected output and input states. A possibility is to take zero time delay for connections. However, to be more general, nonzero delays in connections are allowed as well.

Connection of one agent to the other

$\forall$ A : AGENT $\forall$ B : AGENT $\forall$ C : COMMUNICATION_TYPE $\forall$ I : INFO
[A $\neq$ B] $\Rightarrow$
output(A) : communication_from_to(C, I, S, A, B) $\bullet\!\!\rightarrow\!\!\!\!\rightarrow_{5,5,10,10}$
input(B) : communication_from_to(C, I, S, A, B)

Connection of an agent to the External World

$\forall$ A : AGENT $\forall$ I : INFO
output(A) : to_be_acquired_by(I, A) $\bullet\!\!\rightarrow\!\!\!\!\rightarrow_{5,5,10,10}$
input(EW) : to_be_acquired_by(I, A)

Connection of the External World to an agent

$\forall$ A : AGENT $\forall$ I : INFO $\forall$ S : SIGN
output(EW) : acquired_information_for(I, S, A) $\bullet\!\!\rightarrow\!\!\!\!\rightarrow_{5,5,10,10}$
input(A) : acquired_information_for(I, S, A)

Connection of an agent to the system output state

$\forall$ A : AGENT $\forall$ O : OBJECT
output(A) : conclusion(object_type(O)) $\bullet\!\!\rightarrow\!\!\!\!\rightarrow_{5,5,10,10}$
output(S) : conclusion(object_type(O))

5. Analysis Environment

Apart from an editor to specify dynamic properties, the analysis environment includes two parts; all these tools assume a finite time frame:

1. a tool that, given a set of traces (e.g., generated by simulation based on the component properties or testing of a prototype realisation), checks

any dynamic property of a component-based design expressed in the Temporal Trace Language TTL. In additiona tool has been developed that, given a trace checks for any property expressed in terms of $\bullet\!\!\rightarrow\!\!\!\!\rightarrow$ where exactly in the trace this property fails.

2. a tool that, given an executable specification, for any dynamic property in leads to format proves or disproves whether it is entailed by the dynamic properties in leads to format of the components.

5.1. CHECKING A DYNAMIC PROPERTY AGAINST A SET OF TRACES

This application assumes that at a certain point in time within a design process both a set of requirements for dynamic properties of a design as a whole, and a generated candidate for the design are available. To check whether a required dynamic property for the design (expressed by a temporal formula in TTL with a universally quantified variable $\mathcal{T}$ for a trace) is fulfilled in a given trace or set of traces, a software environment based on some Prolog and C programmes (of about 4000 lines) has been developed. Within this program temporal formulae from TTL are represented by nested term structures based on the logical connectives. Specification of properties is supported by an editor in which properties in terms of TTL can be expressed more or less as written below.

GR1 Successfulness

For any trace of the system, there exists a point in time such that in this trace at that point in time the system will provide the relevant conclusion.
Using the Temporal Trace Language TTL, this is formally expressed by:

$$\forall\, \mathcal{T} : \text{TRACES}\ \exists\, \text{O} : \text{OBJECT}\ \exists\, \text{t}$$
$$\text{state}(\mathcal{T}, \text{t}, \text{output(S)}) \models \text{conclusion(object\_type(O))}$$

This can also be expressed in simple format as follows:

$$\exists\, \text{O} : \text{OBJECT}\quad \text{true} \rightarrow\!\!\!\!\rightarrow_{0,1000,10,10} \text{output(S)} : \text{conclusion(object\_type(O))}$$

GR2 Correctness

For any trace of the system and any point in time, if in this trace the system provides information at t, then this information is correct.
Formally expressed in TTL format by:

$$\forall\, \mathcal{T} : \text{TRACES}\ \ \forall\, \text{O} : \text{OBJECT}\ \forall\, \text{t}$$
$$\text{state}(\mathcal{T}, \text{t}, \text{output(S)}) \models \text{conclusion(object\_type(O))} \Rightarrow$$
$$\text{state}(\mathcal{T}, \text{t}, \text{output(EW)}) \models \text{true\_in\_world(object\_type(O), pos)}$$

This can also be expressed in simple format as follows:

$$\forall\, \text{O} : \text{OBJECT}$$
$$\text{output(W)} : \text{true\_in\_world(object\_type(0), pos)} \bullet\!\!\!\!\leftarrow_{0,1000,10,10}$$
$$\text{output(S)} : \text{conclusion(object\_type(O))}$$

GR3 Conservatism
For any trace of the system and any point in time, if in this trace the system provides a conclusion at t, then for all points in time t' later than t in this trace at t' the system provides the same conclusion.
Formally expressed in TTL format by:

$\forall \mathcal{T}$: TRACES $\forall$ O : OBJECT $\forall$ t
state($\mathcal{T}$, t, output(S)) $\models$ conclusion(object_type(O)) $\Rightarrow$
$\forall$t' > t state($\mathcal{T}$, t', output(S)) $\models$ conclusion(object_type(O))

This can also be expressed in simple format as follows:

$\forall$ O : OBJECT
output(S) : conclusion(object_type(O)) $\twoheadrightarrow_{1,1,1,1}$ output(S) : conclusion(object_type(O))

GR4 Cooperation necessity
For any trace of the system and any point in time, if in this trace agent A provides a conclusion, then at an earlier point in time agent B has communicated to agent A information acquired by B.
Formally expressed in TTL format by:

$\forall \mathcal{T}$: TRACES $\forall$ O1 : OBJECT $\forall$ t
state($\mathcal{T}$, t, output(A)) $\models$ conclusion(object_type(O1)) $\Rightarrow$
$\exists$t' < t $\exists$ O2 : OUTLINE
state($\mathcal{T}$, t', output(B)) $\models$ communication_from_to(world_info, view(B, O2), pos, B, A)

This can also be expressed in simple format as follows:

$\forall$O1:OBJECT $\forall$O2:OUTLINE $\forall$A:AGENT $\forall$B:AGENT
[A $\neq$ B] $\Rightarrow$
output(B) : communication_from_to(world_info, view(B, O2), pos, B, A) $\bullet\!\!-_{0,1000,10,10}$
output(A) : conclusion(object_type(O1))

As an example, the specified successfulness property is represented as a nested term structure internally within the software environment:

```
forall(M, ex(T2, INF,
    holds(state(M, T2, output(S)), communication_from_to(INF, S:SYSTEM, U:USER), true) ) )
```

Traces are represented by sets of Prolog facts of the form

```
holds(state(m1, t(2), input(component)), a)), true).
```

where m1 is the trace name, t(2) time point 2, and a is a state formula in the ontology of the component's input. It is indicated that state formula a is true in the component's input state within the design at time point T2. The Prolog programme for temporal formula checking uses Prolog rules such as

```
sat(and(F,G)) :- sat(F), sat(G).
sat(not(and(F,G))) :- sat(or(not(F), not(G))).
sat(or(F,G)) :- sat(F).
sat(or(F,G)) :- sat(G).
sat(not(or(F,G))) :- sat(and(not(F), not(G))).
```

that reduce the satisfaction of the temporal formula finally to the satisfaction of atomic state formulae at certain time points, which can be read from the trace representation.

Another part of the program takes a trace of a component-based design as input, and specifications of dynamic properties (assumed in executable format) and generates an interpretation of what happens in this trace: it provides a check where exactly these dynamic properties actually hold in that trace.

The program marks any deficiencies in the trace compared with what should be there due to the temporal relationships. If a relationship does not hold completely, this is marked by the program. The program produces yellow marks for unexpected events. At these moments, the event is not produced by any temporal relationship; the event cannot be explained. The red marks indicate that an event has not happened, that should have happened. In addition to checking whether the rules hold, the checker produces an informal reading of the trace. The reading is automatically generated, using a simple substitution, from the information in the given trace.

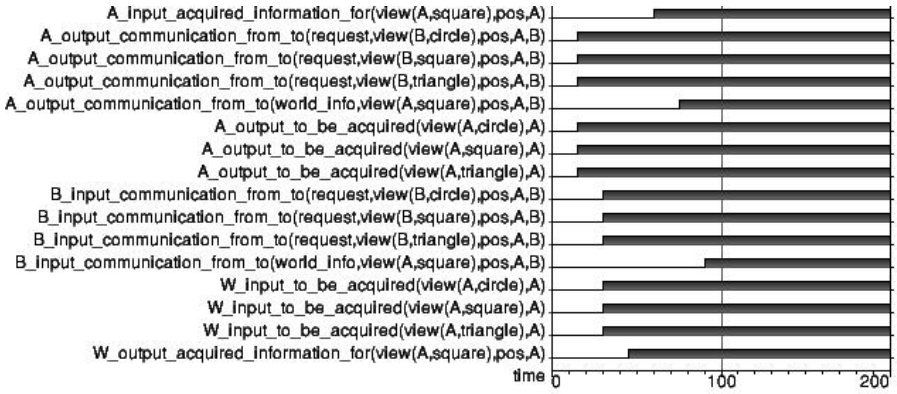

Figure 4. A nonsuccessful trace

This environment can be used to compare any trace (for example obtained by prototyping or simulation) to the possible dynamics of a component-based design. For example, it can be found that a certain part within the realisation does not fulfill the relevant component's specified dynamic properties. As an example, in Figure 4 a trace is considered where agent B is expected to have a certain combination of properties, under which the

information acquisition reactiveness property but actually does not have this property (nor the information acquisition pro-activeness property). No conclusion is reached by the system. What is the cause of this failure? By checking all expected properties of B (and of the other components) it turns out that indeed the information acquisition reactiveness property fails, whereas the other expected properties do not fail. This pinpoints the cause of the failure. The result of the check is (taken from the larger picture in the interface) as depicted in Figure 5. It shows in red which and when outputs were expected from agent B that actually lack in the trace.

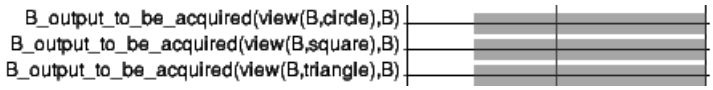

Figure 5. Pinpointing the cause of nonsuccessfulness

For all traces, the successfulness property GR1, the correctness property GR2, the conservatism property GR3, and the cooperatioon necessity property GR4 as specified in simple form above can be checked automatically, and actually have been checked for a number of traces.

As mentioned, to use this software environment, both a specification of a relevant property for the overall design and a set of traces of the considered design are needed. The relevant dynamic property is assumed to result from a requirements analysis during the design process. The set of traces against which the requirement is checked, can be provided in two manners. The first manner to obtain this set of traces is by simulation of the candidate component-based design. Here it is assumed that the (reusable) components used in the component-based design all have associated dynamic properties that hold for them. Moreover, it is assumed that these dynamic properties of the components can be expressed in executable format, based on the relation $\bullet\!\!\rightarrow\!\!\!\!\rightarrow$. If these assumptions are fulfilled, simulation can be performed based on the simulation software environment described in Section 6. The second manner to obtain a set of traces is by prototyping. This assumes that realisations of the components used are available, and that they can easily be configured to obtain a prototype realisation of the whole design. This prototype realisation is used to do some test sessions, resulting in a set of traces that can be checked automatically as described above.

5.2. PROVING A $\bullet\!\!\rightarrow\!\!\!\!\rightarrow$ PROPERTY FROM AN EXECUTABLE SPECIFICATION

Another software tool (about 300 lines of code in Prolog) addresses the proving of dynamic properties (expressed in terms of $\rightarrow\!\!\!\!\rightarrow$, $\bullet\!\!-$ or $\bullet\!\!\rightarrow\!\!\!\!\rightarrow$) of a component-based design as a whole from an (executable) specification of the

dynamic properties of the components without involving specific traces. This dedicated prover exploits the executable nature of the specification and the 'past/current implies future' (cf. (Barringer et al. 1996)) nature of the property to keep complexity limited.

Using this prover, dynamic properties of the component-based design can be checked that hold for all traces, without generating them all (or a sample set of them) by subsequent simulation or prototyping as is needed in the approach described in Section 5.1. The efficiency of finding such a proof strongly depends on the complexity of the specifications of the dynamic properties for the different components. For example, given the executable specification of the software agents, the successfulness property GR1, the correctness property GR2, the conservatism property GR3, and the cooperatioon necessity property GR4 defined above can be (and actually have been) proven from suitable sets of properties of the components.

Also properties can be disproven. For example, for specifications of sets of dynamic properties of the agent components that do not guarantee information exchange between the agents, the property that 'succesfulness of information acquisition of the two agents implies successfulness' can be (and actually has been) proven not to hold. The prover comes up with a trace that is a counter example against this property: Similarly, for agents that are not conclusion pro-active, the property that 'if each of the agents provides the other agent with the information it acquired implies successfulness' has been disproven.

The efficiency of the prover is reasonable. The price that is paid to keep complexity limited is that only properties of the overall design can be proven that can be written in one of the formats $\twoheadrightarrow$, $\bullet\!\!-$or $\bullet\!\!\rightarrow\!\!\!\!\rightarrow$.

6. Simulation Environment

Based on the declarative executable temporal language, a simulation environment has been created to enable the simulation of a component-based design based on specification (in executable format) of dynamic properties of its component, to be used for testing and evaluation purposes; similar to, e.g., (Al-Asaad and Hayes 1995). In this section, first an example of an executable model is discussed. Then the simulation software is introduced. Finally, some of the results of the experiments with the example executable model are discussed. Input for the simulation environment is a set of executable temporal formulae expressed in terms of the leads to relation $\bullet\!\!\rightarrow\!\!\!\!\rightarrow$ (i.e., in the format defined in Section 3). A software environment has been developed which implements the temporal formalisation of the dynamics as specified by an executable model. First the approach is introduced, then the program will be briefly reviewed, after which some of the results are discussed.

6.1. THE SIMULATION SOFTWARE

The simulation determines the consequences of the temporal relationships forwards in time. Remember that α leads to β, is denoted by $P1{:}\alpha \bullet\!\!\rightarrow\!\!\!\!\rightarrow_{e,\,f,\,g,\,h} P2{:}\beta$, where the time delay λ is taken from the interval [e, f]. The duration parameter g denotes the time span that α must minimally hold, and h denotes the duration parameter that β must minimally hold. In order to make simulation efficient, long intervals of results are derived when starting from long intervals. By applying additional conditions (i.e., $e+h \geq f$), the derivation of longer intervals becomes possible, see Section 3, Figure 3. The logical relationships thus avoid unnecessary work for the simulation software.

The delay value λ can either be chosen randomly within the interval [e, f] each time a relationship is used in simulation, or the λ can be a fixed value in this interval. Selecting either a random or fixed λ enables thorough investigation of the consequences of a particular model.

Following the paradigm of executable temporal logic, cf. (Barringer et al. 1996), but extended to real-time intervals, a 8000 line simulation program was written in C++ to automatically generate the consequences of the leads to relationships within the executable specification of dynamic properties of the components within a component-based design. The program is a special purpose tool to derive the results forwards in time, as in executable temporal logic. After a short look at the method of forward derivation, the specification of the derivation rules is presented.

In order to derive the consequences of the temporal relationships within a specific interval of time, a cycle is performed, starting at time 0. For the set of rules the earliest starting time of the antecedent for each rule, for which the consequent does not already hold, is computed. A rule with earliest start time of the antecedent is chosen. This rule is then fired at that time, adding the consequent to the trace. The cycle is restarted, only looking for antecedents at or after the fire time point, as effects are assumed to occur simultaneously or after their causes. This continues until no more rules can be fired, or the fire time is at or after the end time of the simulation interval.

The program reads a specification of temporal rules from a plain text file. The maximum time for derivation is also specified in that file, the interval [0, MaxTime). In order to specify facts about the environment, (periodic) intervals can be given. The functions not(), and(,), and or(,) can be used to make more complex properties from atoms. The properties have and, or and not given in prefix ordering for the program (instead of infix), i.e., a function is given before its arguments, e.g., and(a, b) instead of (a and b). The +(and +) brackets perform concatenation of their contents, in order to construct identifiers from variables and strings. The $\bullet\!\!\rightarrow\!\!\!\!\rightarrow$ relation is specified using LeadsTo, followed by the e, f, g and h values. Note that the program does not use the $\bullet\!\!-$ (originates from) part of the relation as only forward derivation is

performed. First the timing is given, then the variables are quantified. A restriction is often put on the variables. Then the antecedent and consequent are given. For clarity, tokens are displayed boldface, values and identifiers are not. For example, the information acquisition reactiveness property is expressed as follows; here the keywords of the language are in bold:

```
RuleVarLeadsTo delay 5 5 10 10
 Var ForAll A : AGENT
    ForAll B : AGENT NotEqual B : A
    ForAll O : OUTLINE
 EndVar
   +( A _input_ communication_from_to( request , view( A , O ) , pos , B , A ) +)   o->>
   +( A _output_ to_be_acquired( view( A , O ) , A ) +)
```

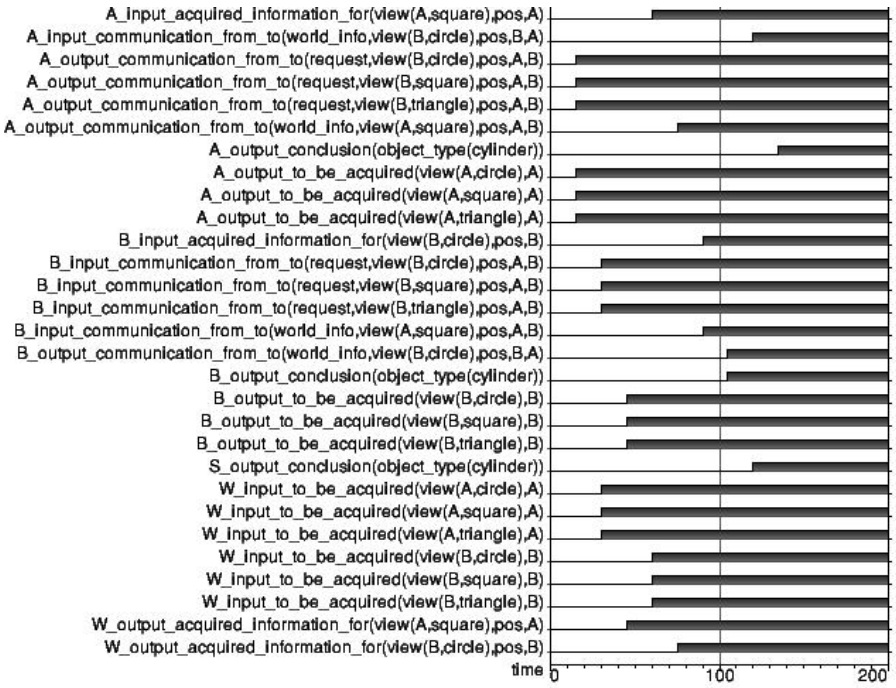

Figure 6. A is fully reactive and proactive;
B is reactive, but proactive in making conclusions.

6.2. SOME SIMULATION RESULTS

Figure 6 shows some of the results of simulation (picture generated by the tools) with the example component-based design. Time is on the horizontal axis. The properties are listed on the vertical axis. The λ is fixed at 0.5. A dark box on top of the line indicates the property is true during that time

period, and a single line indicates that the property is false during that time period. The first line, for example, contains the property that A has as input that what he has acquired is a shape that is no circle. This property is false most of the time, but true from approximately time point 60 on.

Figure 6 shows that the first event is that agent A takes the initiative (around time point 15) to acquire information by itself, and to request B for information. This information is received by B and the External World component around time point 30. The External World provides the information (it is a square) around time point 45; this acquired information is received by A around time point 65. In the meantime, around time point 45 B starts to acquire its own information, which reaches the External World around time point 60. Around time point 75 the External World provides this acquired information for B (it is a circle), which is received by B around time point 90. Agent B drwas the conclusion around time point 105, and also communicates its acquired information to A around this timepoint 105; A receives it around time point 120, and draws a conclusion around time point 130. Figure 7 shows another simulation trace generated by the tools.

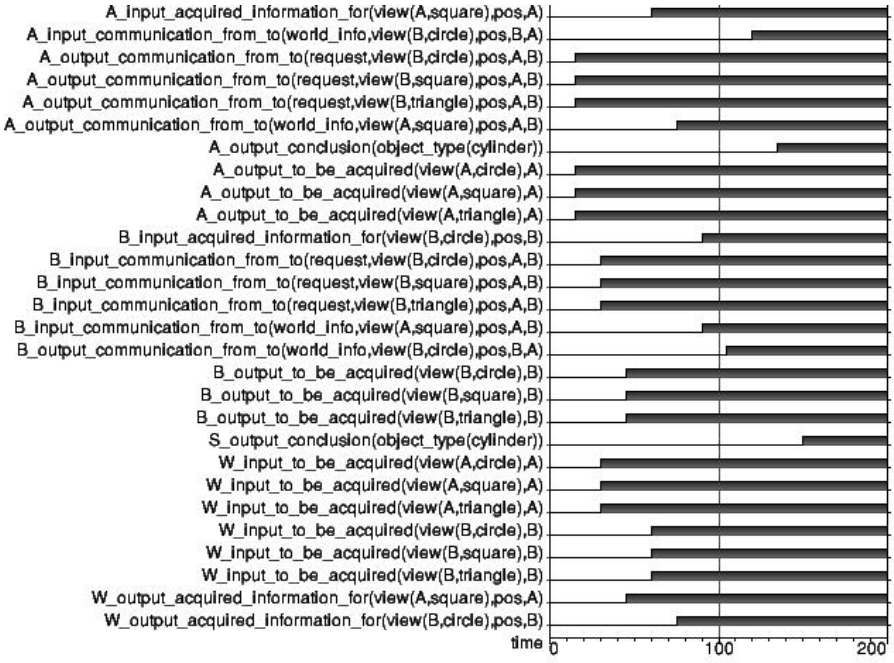

Figure 7. A is fully proactive and reactive, B is reactive only.

7. Discussion

In this paper the use of specified properties of the dynamics of a component-based design in the context of evaluation of the design is addressed. A declarative temporal trace language TTL is offered to specify dynamic properties, and a declarative executable language is defined as a basis for simulation and more efficient analysis. This executable language belongs to the paradigm of executable temporal languages, cf. (Barringer et al. 1996), but adds the use of real-time intervals. Models can be specified in a declarative manner based on a temporal 'leads to' relation which is parameterized by four real valued parameters for time durations and delays. Within the simulation environment such models can be executed.

To specify dynamic properties in a more expressive manner, the language TTL is used: a Temporal Trace Language that belongs to the family of languages to which also situation calculus (McCarthy and P. Hayes 1969; Reiter, 2001), event calculus (Kowalski and Sergot 1986), and fluent calculus (Hölldobler and Thielscher 1990) belong. The executable language for simulations is definable within this much more expressive language. Supporting tools have been implemented that consist of:

- A software environment for simulation of a component-based design on the basis of a specification of dynamic properties of the components and their connections in executable 'leads to' format
- A software environment for evaluation of dynamic properties against traces for a component-based design, both for properties in TTL format and in 'leads to' format
- A software environment to automatically prove (or disprove) 'leads to' properties of the overall design from 'leads to' properties of its components and their connections

The temporal trace language TTL used in our approach is much more expressive than standard or extended modal temporal logics as described, for example, in (Henzinger, Nicollin, Sifakis and Yovine 1994; Bouajjani, Lakhnech and Yovine 1996; Yovine 1997; Fisher 1994; Clarke, Grumberg and Peled 2000; Manna and Pnueli 1995; Stirling 2001), in a number of respects. In the first place, it has *order-sorted predicate logic* expressivity, whereas most standard temporal logics are propositional. Secondly, the explicit reference to *time points and time durations* offers the possibility of modelling the dynamics of real-time phenomena. These first two points apply only partially to logics where it is possible to have real numbers for time and arithmetical operations and order relations to express constraints between time points, as in (Henzinger, Nicollin, Sifakis and Yovine 1994; Bouajjani, Lakhnech and Yovine 1996; Yovine 1997).

Third, the possibility to quantify over traces allows for specification of *more complex dynamics*. As within most temporal logics, reactiveness and

pro-activeness properties can be specified. In addition, in our language also properties expressing different types of adaptive behaviour can be expressed. For example an adaptive property such as 'exercise improves skill', or 'the better the experiences, the higher the trust' (trust monotonicity) which both are a relative property in the sense that it involves the comparison of two alternatives for the history. This type of property can be expressed in our language, whereas in standard forms of temporal logic different alternative histories cannot be compared. The same difference applies to situation calculus, event calculus, fluent calculus, and the languages in (Henzinger, Nicollin, Sifakis and Yovine 1994; Bouajjani, Lakhnech and Yovine 1996; Yovine 1997).

Fourth, in TTL it is possible to define *local languages for parts* of a system. For example, the distinctions between components, and between input and output languages are crucial, and are supported by the language, which also entails the possibility to quantify over system parts and changing system parts over time; for example, this allows for specification of system configuration modification over time; cf. (Dastani, Jonker and Treur 2002)

The approach proposed suggests that (documentation) libraries of reusable components should as much as possible include specifications of dynamic properties in the simpler 'leads to' format. If these properties can be taken from such a library, and requirements on the dynamics of the design as a whole are formally specified as indicated, then the support as described can work quite well. If, however, no specifications of the dynamic properties of the reusable components are known, then as part of the design process these properties and their specifications have to be identified first.

References

Al-Asaad, H. and Hayes, J. P. (1995). Design Verification via Simulation and Automatic Test Pattern Generation, *International Conference on Computer-Aided Design*, 1995, pp. 174-180.

Barringer, H., M. Fisher, D. Gabbay, R. Owens, & M. Reynolds (1996). The Imperative Future: Principles of Executable Temporal Logic, Research Studies Press Ltd. and John Wiley & Sons.

Bouajjani, A., Lakhnech, Y., and Yovine, S. (1996). Model checking for extended timed temporal logic. In *Proc. of the 4th International Symposium Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT'96*, Uppsala, Sweden, September 1996. Lecture Notes in Computer Science, vol. 1135, Springer-Verlag.

Brazier, F.M.T., Jonker, C.M., and Treur, J. (1998). Principles of Compositional Multi-agent System Development. In: J. Cuena (ed.), Proceedings of the 15th IFIP World Computer Congress, WCC' 98, Conference on Information Technology and Knowledge Systems, IT&KNOWS' 98, 1998, pp. 347360. To be published by IOS Press, 2002.

Clarke, E.M., Grumberg, O., and Peled, D.A. (2000). *Model Checking*. MIT Press.

Dastani, M., Jonker, C.M., and Treur, J. (2002). A Requirement Specification Language for Configuration Dynamics of Multi-Agent Systems. In: Wooldridge, M., Weiss, G., and Ciancarini, P. (eds.), *Proc. of the 2nd International Workshop on Agent-Oriented*

Software Engineering, AOSE'01. Lecture Notes in Computer Science, vol. 2222. Springer Verlag. To appear, 2002.

Fisher, M. (1994). A survey of Concurrent METATEM — the language and its applications. In: D.M. Gabbay, H.J. Ohlbach (eds.), *Temporal Logic — Proceedings of the First International Conference*, Lecture Notes in AI, vol. 827, pp. 480–505.

Henzinger, T., Nicollin, X., Sifakis, J., Yovine, S. (1994). Symbolic model checking for real-time systems. Information and Computation, 111(2):193--244, June 1994. Academic Press

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, *Proc. of the 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW'99*. Lecture Notes in AI, vol. 1647, Springer Verlag, 1999, pp. 8-27.

Hölldobler, S., and Thielscher, M. (1990). A new deductive approach to planning. *New Generation Computing*, 8:225-244, 1990.

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, vol. 1536, Springer Verlag, 1998, pp. 350-380. Extended version in: *International Journal of Cooperative Information Systems*. In press, 2002.

Kowalski, R., and Sergot, M. (1986). A logic-based calculus of events. *New Generation Computing*, 4:67-95, 1986.

Manna, Z., and Pnueli, A. (1995). *Temporal Verification of Reactive Systems: Safety*. Springer Verlag.

McCarthy, J. and P. Hayes, P. (1969). Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4:463--502, 1969.

Reiter, R. (2001). *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press, 2001.

Roever, W.P. de, H. Langmaack, A. Pnueli (eds.), *Proceedings of the International Workshop on Compositionality, COMPOS'97*. Lecture Notes in Computer Science, vol. 1536, Springer Verlag, 1998.

Stirling , C. (2001). *Modal and Temporal Properties of Processes*. Springer Verlag.

Yovine, S. (1997). Kronos: A verification tool for real-time systems. *International Journal of Software Tools for Technology Transfer*, vol. 1, pp. 123-133.