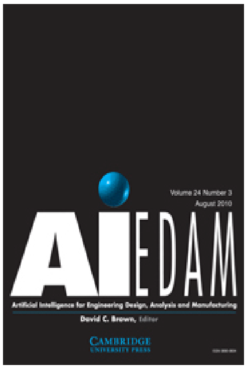

Formal analysis of design process dynamics

Tibor Bosse, Catholijn M. Jonker and Jan Treur

Artificial Intelligence for Engineering Design, Analysis and Manufacturing / Volume 24 / Special Issue 03 / August 2010, pp 397 - 423
DOI: 10.1017/S0890060409990114, Published online: 23 March 2010

Link to this article: http://journals.cambridge.org/abstract_S0890060409990114

How to cite this article:
Tibor Bosse, Catholijn M. Jonker and Jan Treur (2010). Formal analysis of design process dynamics. Artificial Intelligence for Engineering Design, Analysis and Manufacturing, 24, pp 397-423 doi:10.1017/S0890060409990114

Artificial Intelligence for Engineering Design, Analysis and Manufacturing (2010), *24*, 397–423.
© Cambridge University Press, 2010. 0890-0604/10 $25.00
doi:10.1017/S0890060409990114

Formal analysis of design process dynamics

TIBOR BOSSE,[1] CATHOLIJN M. JONKER,[2] AND JAN TREUR[1]

[1]Department of Artificial Intelligence, Vrije Universiteit Amsterdam, Amsterdam, The Netherlands
[2]Department of Mediametics, Delft University of Technology, Delft, The Netherlands

(RECEIVED July 25, 2007; ACCEPTED June 5, 2009)

Abstract

This paper presents a formal analysis of design process dynamics. Such a formal analysis is a prerequisite to come to a formal theory of design and for the development of automated support for the dynamics of design processes. The analysis was geared toward the identification of dynamic design properties at different levels of aggregation. This approach is specifically suitable for component-based design processes. A complicating factor for supporting the design process is that not only the generic properties of design must be specified, but also the language chosen should be rich enough to allow specification of complex properties of the system under design. This requires a language rich enough to operate at these different levels. The Temporal Trace Language used in this paper is suitable for that. The paper shows that the analysis at the level of a design process as a whole and at subprocesses thereof is precise enough to allow for automatic simulation. Simulation allows the modeler to manipulate the specifications of the system under design to better understand the interlevel relationships in his design. The approach is illustrated by an example.

Keywords: Declarative Modeling; Design Processes; Dynamics; Logical Analysis; Simulation

1. INTRODUCTION

Providing automated support to manage the dynamics of a design process is in most cases far from trivial. For example, some of the requirements put forward by Corkill (2000) are that a complete design process representation is needed, with sufficient detail to allow for direct execution. Brown and Chandrasekaran (1989), Heller and Westfechtel (2003), Baldwin and Chung (1995), and Gero and Kannengiesser (2006) also stated that supporting the management of the dynamics of a design process is an important challenge to be addressed. The current paper aims to provide a formal representation of a complete design process in sufficient detail to allow for automatic simulation of the process.

The type of design considered is the design of component-based (e.g., software) systems for dynamic applications. It allows the (re)use of components for which the (dynamic) properties are known. The required overall dynamics is obtained by the correct combination of a number of such components. Finding a correct combination is not straightforward, in particular with respect to the dynamics in the overall system in relation to the available reusable components and their dynamic properties. This aspect especially motivates the goal of this paper to formally analyze the design process and provide an automated simulation of the design process. The simulation presented here is still a long way off from being a design support system, because the whole aspect of the usability of the system and user interaction are not addressed. This paper contributes to such a design support system by providing the fundamental analysis up to the level where automatic simulation can be performed. The complexity of the intended analysis is high, as the following main tasks of a design process are by themselves complex:

1. maintaining the property specifications of (reusable) components,
2. maintaining the requirements on the overall system to be designed (usually in close contact with a representative of the party that asked for the design process: a stakeholder),
3. refinement of requirements to more specific requirements (usually in cooperation with the stakeholder),
4. revision of requirements on the basis of the process thus far (usually in cooperation with the stakeholder),
5. determination of proper reusable components on the basis of their properties in order to find (a description of) a component-based system that satisfies the requirements,

Reprint requests to: Tibor Bosse, Department of Artificial Intelligence, Vrije Universiteit Amsterdam, De Boelelaan 1081a, Amsterdam 1081 HV, The Netherlands. E-mail: tbosse@cs.vu.nl

6. checking whether [a design object description (DOD) of] a component-based system with known properties of its components satisfies the requirements,
7. revision of a DOD that does not fully satisfy the requirements, and
8. coordination of the different processes within the design task.

Some of these tasks only concern requirements specification and specification and evaluation of dynamic properties of *DODs*, in particular, the first, second, and sixth task. These tasks abstract from the dynamics of the actual design process as a whole; they have been addressed in Jonker et al. (2002). The other tasks essentially deal not with (required) dynamics of design *objects* but with the dynamics of *design as a process*. The analysis of this *design process dynamics* is the subject of the current paper.

During a design process, two important concepts play a role: a design problem statement and a solution specification. A *design problem statement* consists of

- a set of requirements in the form of dynamic properties on the overall system behavior that have to be fulfilled;
- a partial description of (prescribed) system architecture that has to be incorporated; and
- a partial description of (prescribed) dynamic properties of elements of the system that have to be incorporated, for example, for components, transfers, parts, and interactions between parts.

A *solution specification* for a design problem is a specification of a design object (both structure and behavior) that fulfills the imposed requirements on overall behavior, and includes the given (prescribed) descriptions of structure and behavior. Here "fulfilling" the overall behavior requirements means that they are implied by the dynamic properties for components, transfers, and interactions between parts within the specification.

The approach used for the formal analysis emphasizes the difference between the generic aspects of the design process and the specific aspects of a system to be designed during the process. The logical specification of the generic aspects refers to such generic concepts as the system requirements and DODs. As a result, we provide a multilevel specification: the generic level and the example specific level of the system to be designed. Note that the specific level can contain various complex specifications as well, corresponding to the complexity of the system to be designed.

This paper is organized as follows. Section 2 discusses a formalization of design process dynamics in terms of design states and design traces. Section 3 addresses some dynamic properties of design processes. Section 4 gives an overview of an example design process. A relevant global requirement for the example system to be designed is given in Section 5. This global requirement for the overall system is refined to local requirements for parts of the system. In Section 6, based on the adopted design problem, an example design trace is discussed. After that, Section 7 describes a simulation model of the example design process, whereas Section 8 discusses some example simulation traces. In Section 9 the example design process is analyzed in terms of dynamic properties. In particular, it discusses results of automated checks of these dynamic properties against the example simulation traces discussed in Section 8. Section 10 presents some of the logical relationships between these dynamic properties. Finally, Section 11 is a discussion.

2. DESCRIBING DESIGN PROCESS DYNAMICS

The analysis of the dynamics of a design process requires a formalization of the concepts involved. The formalization introduced in this section is based on Treur (1991) and Jonker and Treur (2002), which identified such fundamental concepts as design state and design trace, where design states and design traces are composed of a part for requirements and a part for DODs, an approach adopted from Treur (1991). More information can be found in Gavrila and Treur (1994) and Brazier et al. (1994, 1996).

2.1. States of a design process

The state of a design process at a certain time point is described as a combined design state consisting of two states, $S = \langle S_1, S_2\rangle$ where S_1 is the *requirements state* (including the current requirements set) and S_2 is the *DOD state* (including the current DOD).

A particular design process shows a sequence of states of requirements sets and of DODs over time. A *design state ontology* Ont includes ontology for DODs and for requirements. The set of *ground state atoms* over Ont is denoted by GSTATOMS(Ont). A *design state S* over a design state ontology Ont (including ontology for design objects and requirements) is a mapping assigning truth values to the ground atoms S: GSTATOMS(Ont) → {true, false, undefined}. The set of all possible states over Ont is denoted by STATES(Ont).

2.2. Traces of a design process

Design traces are time-indexed sequences of such design states. To describe such sequences a fixed *time frame* T is assumed, which is linearly ordered (e.g., the real or natural numbers). A *trace* γ over a design state ontology Ont and time frame T is a mapping γ: $T \rightarrow$ STATES(Ont), that is, a sequence of states γ_t ($t \in T$) in STATES(Ont). The set of all traces over state ontology Ont is denoted by TRACES(Ont). Depending on the application, the time frame T 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.

3. DYNAMIC PROPERTIES OF DESIGN PROCESSES

Specification of dynamic properties of a design process 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. Second, specification of dynamic properties of a process can be done to *analyze* its dynamics, for example, to find out how certain properties of a design process as a whole relate to properties of a certain subprocess, or to verify or test a design model.

3.1. Specifying dynamic properties of a design process

To formally specify dynamic properties that express characteristics of dynamic processes (such as design) from a temporal perspective an expressive language is needed. To this end the *Temporal Trace Language* (TTL) is used as a tool (cf. Jonker & Treur, 2002; Bosse et al., 2006*a*). This language can be classified as a predicate logic-based reified temporal language; see Galton (2003, 2006). Other classes of temporal languages are the modal temporal logics such in Barringer et al. (1996), Goldblatt (1992), and Fisher (2005). The expressivity of these languages (within a world) is usually limited to propositional logic. TTL is briefly defined in the following.

3.1.1. TTL for dynamic properties

To start, an order-sorted predicate logic ontology Ont to describe state properties is assumed, consisting of sorts, subsort relations, constants in sorts, and functions and relations over sorts (e.g., Manzano, 1996). Among these sorts are also sorts for real and integer numbers, and among the functions the standard arithmetical functions. This makes TTL a *hybrid declarative language*. Moreover, in TTL traces γ, time points t and state properties p can be used as first class citizens in sorts TRACES, TIME, and STATPROP, respectively. The set of *dynamic properties* DYNPROP(Ont) is the set of temporal statements that can be formulated with respect to traces based on the state ontology Ont in the following manner. Given a trace γ over state ontology Ont, a certain state during a design process at time point t is denoted by $\mathsf{state}(\gamma, t)$. These states can be related to state properties via the formally defined satisfaction relation denoted by the infix predicate $|=$. Here $\mathsf{state}(\gamma, t) |= p$ denotes that state property p holds in trace γ at time t. This predicate is comparable to the Holds-predicate in the situation calculus and event calculus (cf. Kowalski & Sergot, 1986; Reiter, 2001). Notice that here state properties are represented (reified) by terms to denote them as objects in TTL. Based on these statements, dynamic properties can be formulated in a formal manner in a sorted first-order predicate logic with sorts TIME for time points, TRACES for traces, and STATPROP for state formulae, using quantifiers and ordering relations over time and the usual first-order logical connectives such as $\neg$, &, $\vee$, $\Rightarrow$, $\forall$, and $\exists$.

Because TTL uses order-sorted predicate logic as a point of departure, it inherits the standard semantics of this variant of predicate logic. That is, the semantics of TTL is defined in a standard way, by interpretation of sorts, constants, functions and predicates, and a variable assignment. In addition the semantics involves some specialized aspects. A number of standard sorts are present, so the elements of these sorts are limited to instances of specified terms in these sorts as usual, for example, in logic programming semantics. For example, for the sort TIME it is assumed that in its semantics its elements consist of the time points of the fixed time frame chosen. Moreover, for the sort TRACE, it is assumed that in its semantics its elements consist of a (limited) number of traces named by constants. Furthermore, for the sort STATPROP for state properties it is assumed that in its semantics its elements consist of the set of terms denoting the propositions built in a chosen state language (this is called reification). A full description of the technical details of TTL's semantics is beyond the scope of the current paper. For this purpose, see Sharpanskykh and Treur (2005).

3.1.2. The language LEADSTO for executable dynamic properties

To be able to perform some automated experiments with design processes, the full expressivity of TTL is not required. Therefore, a simpler temporal logical language to specify simulation models has been used for this purpose. This language LEADSTO (Bosse et al., 2007), which is a sublanguage of TTL, enables to model direct temporal dependencies between two state properties in successive states, as occur in specifications of a simulation model (e.g., if in the current state, state property p holds, then in the next state, state property q holds). This language is executable, and therefore enables the automatic generation of simulated traces; for other executable temporal languages based on modal logic, see Barringer et al. (1996). This section briefly introduces the logical format used for these LEADSTO simulation models. This executable format is defined as follows. Let α and β be state properties of the form "conjunction of atoms or negations of atoms." In the LEADSTO language the notation $\alpha \twoheadrightarrow_{e,f,g,h} \beta$, means

IF: state property α holds for a certain time interval with duration g

THEN: after some delay (between e and f) state property β will hold for a certain time interval of length h

Note that within the atoms free variables may occur, which are considered universally quantified over the whole formula. Moreover, numerical functions may be used to perform calculations. In this way, LEADSTO allows us to represent and execute difference/differential equations with arbitrary (and possibly dynamic) step size (see also Bosse et al., 2008). For the complete syntax and semantics of LEADSTO, see Bosse et al. (2007). For a formal definition of the LEADSTO language in terms (as a sublanguage) of the language TTL, see Jonker et al. (2003) and Bosse et al. (2006*a*).

3.2. Dynamic properties at different levels of aggregation

Based on their different levels of aggregation (or by considering in how far they cover the process as a whole or only part of the process), two different types of dynamic properties can be distinguished: local properties and global properties.

3.2.1. Local properties

Local properties only concern the smallest steps (taken into account in the conceptualization of the process) in the process under analysis. An example local property of a design process might be (simplified, and in semiformal notation) the following: at every point in time,

IF: a requirement **r** is imposed on the object to be designed
AND: this requirement can be refined into subrequirement **q**
THEN: at the next point in time, subrequirement **q** will be imposed on the object to be designed

3.2.2. Global properties

In contrast, a global property is a nonlocal property that concerns the overall process (taken into account) in the process under analysis. An example is

Eventually there is a committed requirement set **R** and
a DOD **D** such that, for each requirement **r** in **R**,
the DOD **D** satisfies requirement **r**

Note that often, but not always, local properties can be represented as executable dynamic properties (e.g., using the LEADSTO format introduced above). As a result, these executable local properties can be used to generate simulation traces. Instead, global properties often have a higher complexity, for example, because they relate states at multiple time points to each other in a manner that is beyond a simple causal relationship (e.g., "if *X* and later *Y* then earlier *Z*," or "eventually *X* or *Y* happens"). Therefore, the more expressive TTL is needed to formalize these global properties. In the approach taken in this paper, executable local properties are used to generate simulation traces, and global properties are used to analyze these simulation traces, that is, to check (using automated tools) whether they show the expected behavior. More complex local and global dynamic properties for design processes and their formalizations will be presented in subsequent sections.

4. AN EXAMPLE DESIGN PROCESS

To address in more detail the analysis of design process dynamics, an example design process was taken. The analysis approach is described and evaluated for this example design process. The example design process concerns the design of a cooperative information gathering agent system (see Section 4.2). The design approach is by requirements refinement (see Section 4.1).

4.1. Designing by requirements refinement

A design process of a complex system (e.g., a software system) usually starts by specifying requirements for the overall system behavior. They express the dynamic properties that should "emerge" if appropriate components are designed and combined in a proper manner. Given these requirements on overall system behavior, the system is designed in such a manner that the requirements are fulfilled.

Between dynamic properties at different levels of aggregation within a complex system (to be) designed, certain interlevel relationships can be identified; overall behavior of the design object can be related to dynamic properties of parts of the design object and properties of interaction between these parts via the following pattern:

dynamic properties for the parts
& dynamic properties for interaction between parts
⇒ dynamic properties for the design object

Thus, if for a design problem, requirements in the form of dynamic properties for the overall system behavior are given, this scheme shows that to fulfill these overall dynamic properties, dynamic properties for certain parts and for interaction between these parts are needed that together imply the overall behavior requirements. The process to identify new, refined requirements for behavior of parts of the system and their interaction is called *requirements refinement*. Subsequently, the required dynamic properties of parts can be refined to dynamic properties of certain components and transfers, making use of the pattern:

dynamic properties for components
& dynamic properties for transfer between components
⇒ dynamic properties for a part

4.2. An example design problem

As a case study, the process of designing a multiagent system for cooperative information gathering (Jonker & Treur, 2002) will be analysed in more detail. To get the idea, assume the system to be designed has to consist of three agents: *A*, *B*, and *C* (see Fig. 1). The resulting behavior of the system must be as follows: agent *A* and *B* are able to do some investigations and make up a report on some topic, and communicate that to the third agent *C*. Both *A* and *B* have access to useful

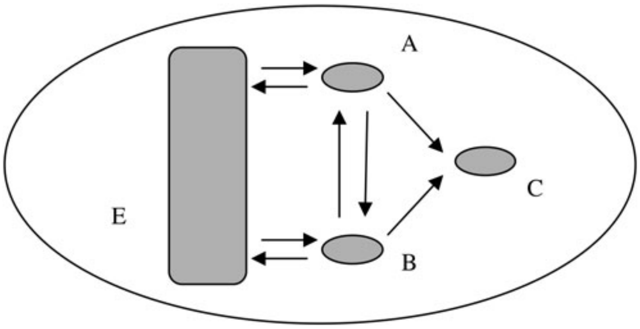

Fig. 1. The example agent system to be designed.

sources of information, but this differs for the two agents. By cooperation they can benefit from the exchange of information that is only accessible to the other agent. If both types of information are combined, conclusions can be drawn that would not have been achievable for each of the agents separately. Why could such a cooperation fail? One of the agents, say *A*, may not be proactive in its individual search for information. This might be compensated if the agent *B* is proactive in asking the other agent for information, but then at least *A* has to be reactive (and not entirely inactive in information search). Some other reasons for failure are one of the agents may not be willing to share its acquired information, or none of them is able or willing to combine different types of information and deduce new conclusions. Thus, agent properties such as *information acquisition proactiveness* and *conclusion proactiveness* could be desirable requirements for parts of the system to be designed.

To make the example more precise: the example agent model is composed of three components: two information gathering agents *A* and *B*, agent *C*, and environment component *E* representing the external world (see Fig. 1). In this figure the ovals denote the three agents. The arrows depict channels for flow of information (so-called information links). Each of the agents is able to acquire partial information from an external source by initiated observations. Initiated observations are modeled by an arrow from the agent to *E*, transferring information on what is to be observed, and by an arrow back transferring information on the results of the observation. For communication the arrows (information links) between the agents are used.

For reasons of presentation, this by itself quite common situation for cooperative information agents is materialized 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 only the side view. By exchanging and combining observation information they are able to classify the object. An agent may be able to draw a conclusion on the object type if it has input on the two views on the object, in the sense that, for example, if the agent knows that the views are a circle and a square, it is concluded that the object is a cylinder.

In most multiagent systems it is common that each agent has its own characteristics or attitudes. In the current system to be designed, the agents used as components in the design can differ in their attitudes toward observation and communication: an agent may or may not be *proactive*, in the sense that it takes the initiative with respect to one or more of the following:

- perform observations,
- communicate its own observation results to the other agent,
- ask the other agent for its observation results, and
- draw conclusions about the classification of the object.

Moreover, an agent may be *reactive* to the other agent in the sense that it responds to a request for observation information

- by communicating its observation result as soon as they are available and
- by starting to observe for the other agent.

The successfulness of the system to be designed will depend on the combination of attitudes of the agents. For example, if both agents are proactive and reactive in all respects, then they can easily come to a conclusion. However, it is also possible that one of the agents is only reactive, and still the other agent comes to a conclusion. Alternatively, an agent that is only reactive in reasoning and in information acquisition may come to a conclusion because of proactiveness of the other agent. Thus, successfulness can be achieved in many ways and depends on subtle interactions between proactiveness and reactiveness attitudes of both agents. The analysis of the example in the following section provides a detailed picture of these possibilities.

5. REQUIREMENTS OF THE EXAMPLE DESIGN PROBLEM

In this section the example agent system to be designed as discussed in the previous section is further elaborated in terms of relevant requirements. Section 5.1 presents the design problem statement, consisting of the requirements on the overall agent system behavior, which includes the dynamic properties for transfer, and the dynamic properties for the environment. Section 5.2 describes a number of variants of a systematic design process (by requirements refinement) to obtain one or more design solutions.

5.1. Design problem statement

The design problem statement of this agent system design problem consists of the overall agent system behavior requirement, interaction dynamics (transfers), and prescribed behaviors for the component *E*. The main requirement imposed on the current agent system is whether or not a result will be generated. This requirement is called DOD global property (DODGP).

DODGP successfulness: For any trace of the system, there exists a point in time such that in this trace at that point in time agent *C* will receive a correct conclusion, either from *A* or from *B* (or from both).

As part of the design problem statement, the behavior of *E* is prescribed by the following environment property for each agent *X* from the set {agent *A*, agent *B*}.

DODEP(X) information provision effectiveness:

IF: *E* receives an information acquisition initiation by *X*
THEN: *E* will generate the correct relevant information for *X*

Furthermore, the behavior about information transfer between agents is prescribed by the following transfer property for several combinations of components *X* and *Y* from the set {agent *A*, agent *B*, agent *C*, external world *E*}.

DODTP(X, Y) information transfer:

IF: X generates information for Y
THEN: Y will receive this information

Thus, it is prescribed that all information generated by an agent for another agent (but no other information) is automatically transferred, without any time duration.

5.2. Design process: Refining requirements

In virtue of which combination of dynamic properties of the agents can success be achieved? In other words, which dynamic properties for the agents imply the property successfulness? How can the requirement on the overall agent system behavior be refined to requirements on agent behaviors? Such a requirements refinement process can be managed more effectively if the overall requirements are not directly related to agent behavior requirements, but one or more intermediate levels are created. The idea is that for the agent system to be successful it is necessary that

- both information sources within the environment E are addressed;
- if they are addressed, they provide the relevant information; and
- if the relevant information is provided by the information sources, a conclusion is drawn.

This first requirements refinement (see top level of Fig. 2) provides the dynamic properties DODGP1, DODGP2, and DODGP3:

DODGP1 information request effectiveness:

At some points in time A and B will start information acquisition to E

DODGP2 information source effectiveness:

IF: at some points in time A and B start information acquisition to E
THEN: E will generate all the correct relevant information for both

DODGP3 concluding effectiveness:

IF: at some points in time E generates all the correct relevant information
THEN: C will receive a correct conclusion

These properties are logically related to DODGP (see also Fig. 2) by the implication

$$\text{DODGP1} \,\&\, \text{DODGP2} \,\&\, \text{DODGP3} \Rightarrow \text{DODGP}$$

A next step in the requirements refinement process is to relate each of the dynamic properties DODGP1, DODGP2, and DODGP3 to agent behavior properties. The complete refinement of these properties is elaborated in Appendix A. In the following, we only present the tree with logical relationships between dynamic properties, without showing the exact definitions of the properties.

In Figure 2, in the form of and AND/OR tree an overview is shown of all possible refinements as discussed. Here X and Y are variables over the set $\{\text{agent } A, \text{ agent } B\}$, where $X \neq Y$. Connections between boxes indicate upward logical implications, and the circles indicates labels of the implications (which we call *branches*), in such a way that the combination of properties below the circle (logically) entail the property above the circle. For example, property DODGP1, DODGP2, and DODGP3 imply (via branch B11) property DODGP. Similarly, property DODBP1(A) and DODBP1(B) imply (via branch B1) property DODGP1. Note that the different alternative branches are indicated by nodes labeled B1 to B5, and that the labels B11 to B16 indicate branches in situations when there are no alternatives. Conjunctions of branches are indicated by arcs connecting them.

Complicated as it may look at first sight, the tree depicted in Figure 2 has been constructed by hand, using natural deduction techniques. Although the full proofs of all interlevel relationships are beyond the scope of this paper, some proof sketches are provided in Appendix A. More background information is also provided in (Jonker & Treur, 2002). To consider a simple (informal) example, consider the interlevel relationship indicated by B1. This states that the properties DODBP1(A) and DODBP1(B) imply DODGP1. The first two properties denote, respectively that "agent A is information acquisition proactive" and that "agent B is information acquisition proactive" (see Appendix A). Assuming these two properties already satisfies property DODGP1, namely, that "At some points in time A and B will start information acquisition to E," which is nothing more that the conjunction of the two.

6. AN EXAMPLE DESIGN TRACE

To illustrate how such a design process works, for the design problem discussed in the previous sections, a simple example design trace is presented (in an informal format) in Table 1. The purpose of this trace is to demonstrate which subsequent steps may be performed for such a design process. In the next sections, a simulation model will be presented that is able to automatically generate such traces.

The subsequent steps shown in Table 1 involve the following activities:

1. Determine what the initial, global design objectives are. Usually, this is something like "designing a certain object X," which is defined in cooperation with a stakeholder.
2. Specify which global requirements need to be fulfilled to reach the design objectives. These requirements are usually domain dependent. For example, in case the

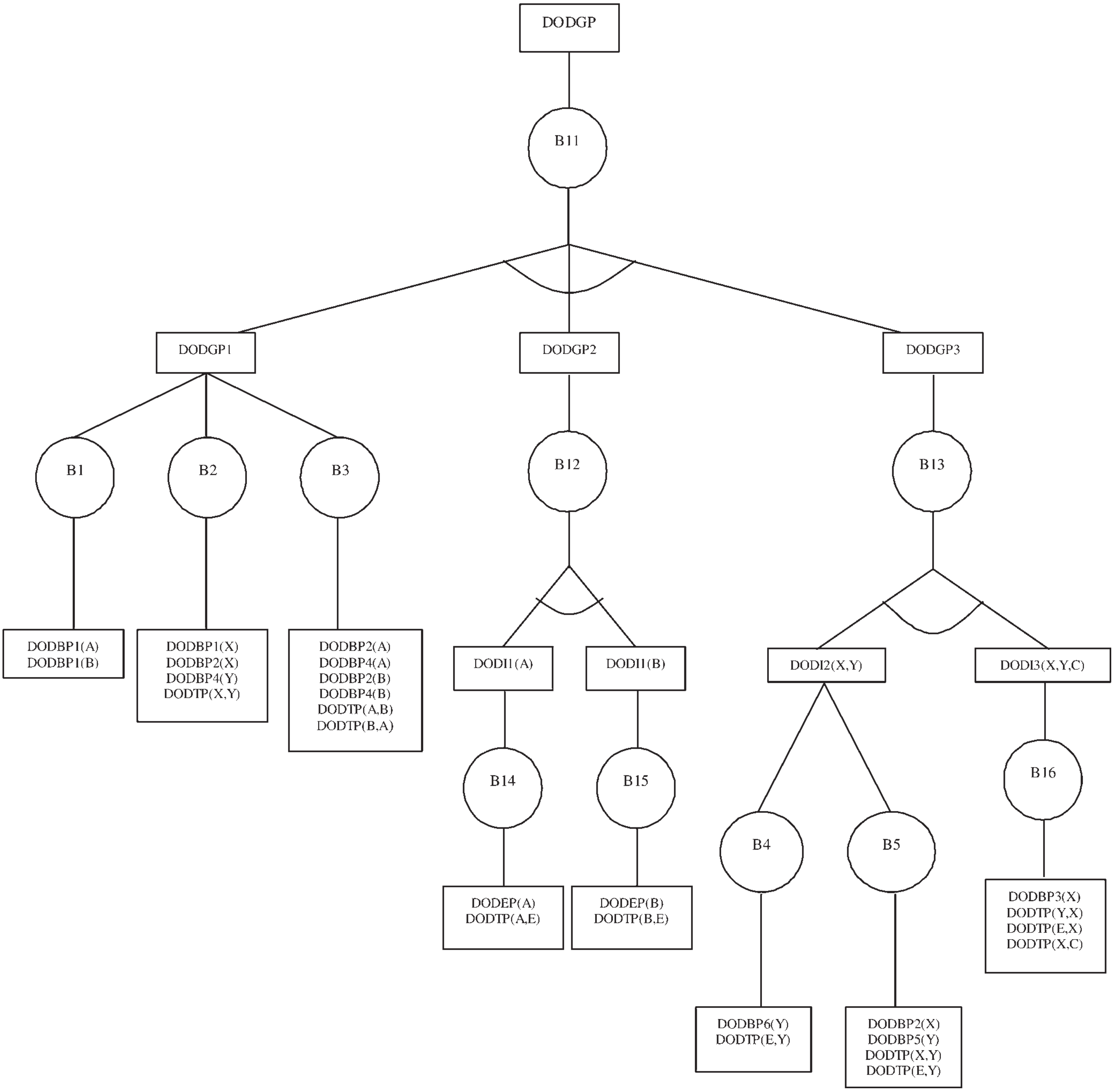

Fig. 2. The overview of all possible requirement refinements.

object to be designed is an information gathering system, then, obviously, one of the requirements it that it should be successful in gathering the desired information. In addition, nonfunctional requirements may be included here, such as “the design process should be finished within *n* months,” or “the design process should not cost more than *c* Euros.”

3. Refine the global requirements (about the design object as a whole) into more detailed requirements (about components and interactions between components), based on refinement knowledge. Such refinement knowledge may be based on strict logical relationships between dynamic properties that are known (such as the tree shown in Fig. 2), or may be of a more heuristic nature. The refinement process may have an iterative character, taking numerous aggregation levels of the tree of logical relationships into account, until the leaves of the tree are reached. For example, in case of the AND/OR tree shown in Figure 2, a set of local requirements is obtained by following all AND alternatives and selecting one OR alternative for each branch. In case of OR branches, additional background knowledge (e.g., about the expected costs or time to fulfill the branch) may be used to make an appropriate choice.

Table 1. *Example design trace*

Step	Informal Description	Effect
1	Initial design process objectives are given.	"A design solution for the example design problem has to be found."
2	Initial requirements are identified. A set of requirements is created, which initially only consists of one requirement: DODGP.	Requirement set (1): {DODGP}
3	Using refinement knowledge, the requirements are refined into requirements for components and interactions (transfers) between components, resulting in a more refined set of requirements. Refinement knowledge, for example, can be based on strict logical relationships between dynamic properties but it can also be of a more heuristic nature.	Requirement set (2): {DODBP1(A), DODBP1(B), DODBP3(X), DODBP6(Y), DODEP(A), DODEP(B), DODTP(A,E), DODTP(B,E), DODTP(Y,X), DODTP(E,X), DODTP(E,Y), DODTP(X,C)}
4	Components (from the library) are identified that satisfy the component requirements obtained, and they are included in the initial DOD state, called DOD(1). The library indicates for each component which of the properties it has [e.g., component C2A satisfies property DODBP1(A)].	DOD(1): {C2A, C2B, C3A, EW}
5	The connections are made according to the transfer requirements and included in the DOD state, thereby creating the next version of the DOD, called DOD(2). This can also make use of the library, or it can be done in a standard manner.	DOD(2): {C2A, C2B, C3A, EW, L1, L2, L3}
6	It is evaluated whether the overall requirements hold for the DOD, based on the logical relationships.	"All requirements evaluated"

Note: DODGP, design object description global property (see Section 5.1).

4. As soon as the lowest level requirements have been identified (which usually are requirements about basic components, or about connections between components), select basic components that satisfy these requirements. Often, such standard components are available in a library (which is assumed in the example used in the next sections). Otherwise, they have to be created.
5. To satisfy lowest level requirements that address transfer between components, select standard components for transfer (e.g., certain predefined communication channels, or information links). In case a certain component can directly communicate with another component, no communication channel is needed at all, and the transfer requirement is automatically satisfied. The total set of all components and "links" makes up the final DOD.
6. Once the final DOD has been established, evaluate whether it indeed satisfies all requirements. Again, this can be done based on knowledge that relates higher level requirements to lower level requirements (see the next section for a discussion about this): once all lower level requirements have been fulfilled, also the higher level requirements have been fulfilled.

In Table 1, the names DODGP, DODBP1(A), and so forth indicate requirements, similar to the ones introduced in the previous section (and of which the formal descriptions are provided in Appendix A); C2A, C2B, EW, and so forth indicate available components that are used in the design process; and L1, L2, and L3 indicate connections or links between components. Examples of such components may be "a piece of memory," "a component for communication," or "a reasoning engine." For more concrete examples for the case study addressed, see Jonker and Treur (2002).

7. A SIMULATION MODEL FOR THE EXAMPLE DESIGN PROCESS

Making use of the formal approach described in the previous section, the dynamics of an example design process can be simulated. This particular example concerns the design of the agent system for cooperative information gathering presented earlier. To be able to simulate the dynamics of such a design, several kinds of domain-specific information (in particular, the logical relationships shown in Fig. 2 and the characteristics of components as stored in the library) have been modeled by means of *sorts* and *facts*. The domain-independent information (e.g., rules that refine a requirement to its subrequirements) has been modeled by means of *local properties* in an executable format. However, notice that one of these local properties is domain specific, namely, the initialization property LP0.

7.1. Sorts

Sorts are used to define all constants that are used within the simulation, and to distinguish different types of constants from each other. Our example contains six sorts: property, nonlocalproperty, localproperty, branch, component, and DOD. Note that the "links," the connections between components, are also modeled as components. Some examples of objects or terms within sorts are

- property: DODGP, DODGP1,…, DODI1(A), DODI1(B),…, DODBP1(X),…
- nonlocalproperty: DODGP, DODGP1,…, DODI1(A), DODI1(B),…
- localproperty: DODBP1(X),…

- branch: B1, B2, B3, B4, B5, B11, B12, B13, B14, B15, B16
- component: C1A, C1B, C2A, C2B, C3A, C3B, C4A, C4B, C5A, C5B, EW, L1, L2, L3, L4
- DOD: DOD(1), DOD(2), DOD(3),...

7.2. Facts

Facts are used to express knowledge that is true during the whole simulation process. The first set of facts represents the logical relationships of Figure 2 in a formal notation. For example, is_a_subrequirement_of_via(DODBP1(A), DODGP1, B1) expresses that property DODBP1(A) is a subrequirement of property DODGP1 via branch B1 (see the lower left edge in Fig. 2). Another example of such a fact representing the logical relationships between requirements is

is_a_subrequirement_of_via(DODGP2, DODGP, B11)

Note that in the latter case there is just one possible branch to choose.

During simulation, these logical relationships are used as heuristics to guide the refinement process. Note that in this example, the tree of logical relationships is complete: it covers all possible combinations of local requirements that together satisfy global property DODGP. However, in more realistic situations this is very unlikely to be the case.[1] Often the logical relationships between requirements that are known beforehand are erroneous and incomplete. Therefore, it is useful to perform an additional evaluation of the resulting DOD at the end of the design process. To perform such an evaluation, a similar tree as in Figure 2 is used, but this time the information is complete and bottom up.[2] It is represented by the following type of facts:

is_implied_by(DODGP, [DODGP1, DODGP2, DODGP3])

In our simplified example, however, the information used for the evaluation fully corresponds to the information represented by the relation is_a_subrequirement_of_via(. . .).

The last set of facts represents the characteristics of the library components. For instance, the fact that component C1A satisfies requirement DODBP1(A) is denoted by the fact holds_for(DODBP1(A), C1A). The fact that component C1A's costs are 500 is denoted by costs(C1A, 500). Moreover, some additional information has been included, such as a quality factor for each component, and the predicted costs for each branch. These kinds of information may be used as heuristic knowledge in the refinement process (see the next section).

holds_for(DODBP1(A), C1A)	costs(C1A, 500)
holds_for(DODBP2(A), C1A)	costs(C1B, 501)
holds_for(DODBP3(A), C1A)	costs(EW, 0)
holds_for(DODBP4(A), C1A)	
quality(C1A, 100)	predicted_costs(B1, 60)
quality(C1B, 100)	predicted_costs(B2, 300)
quality(EW, 1000)	predicted_costs(B3, 900)

7.3. Local properties

As mentioned earlier, local properties are used to model the domain-independent dynamics of the design process. Three types of local properties are distinguished: those that model the dynamics of requirements states, and those that model the dynamics of the DOD states. In this section, only a subset of the local properties used for the simulation is shown. For simplicity, the timing parameters e, f, g, and h have been left out here. The complete specification of the simulation is shown in Appendix B. The local properties shown below make use of the state ontology shown in Table 2.

7.3.1. Properties concerning requirements

Within the process requirements are determined and refined. This process takes into account whether the stakeholder asserts that certain requirements are undesirable.

LP0 initialization. The first local property LP0 expresses that the initial requirements for the system are DODGP and DODCHEAP. Note that, if desired, the user can modify this property by choosing different initial requirements. Formalization:

$$\begin{aligned}&\text{start} \twoheadrightarrow \text{is\_a\_current\_requirement(DODGP)}\\&\quad\wedge \text{is\_a\_current\_requirement(DODCHEAP)}\end{aligned}$$

LP2 undesirable branch determination. These two local properties are used to determine which branches are undesirable. There are two cases: a requirement that belongs to it is undesirable and its total costs are higher than predicted, whereas the requirement DODCHEAP is present. Formalization:

$$\begin{aligned}&\text{is\_a\_subrequirement\_of\_via(p, n, b)}\\&\quad\wedge \text{undesirable\_requirement(p)} \twoheadrightarrow \text{undesirable\_branch(b)}\\&\text{is\_a\_current\_requirement(DODCHEAP)}\\&\quad\wedge \text{total\_branch\_costs(b, x)} \wedge \text{predicted\_costs(b, y)}\\&\quad\wedge \text{x} > \text{y} \twoheadrightarrow \text{undesirable\_branch(b)}\end{aligned}$$

LP4 requirement refinement. Local property LP4 expresses that, if currently a requirement p exists that can be

[1] The only reason why we know in the current example that the tree is complete, is that all possible local behaviors of a component (such as DODBP1, "being information acquisition proactive") are given beforehand in the description of Section 4.2. As a result, by common sense reasoning, one can explore all different combinations of such behaviors and construct a complete tree of combinations that satisfy global property DODGP (e.g., it is quite easy to see that if all agents are only information acquisition reactive, and none of them is request proactive, the resulting system will not be successful).

[2] As opposed to the situation at the start of a design process, after the design process it is much easier to construct a "complete" tree of logical relationships between the overall desired property and the local properties of the components, because the components have been selected. As a result, these relationships can be used to evaluate the behavior of the designed DOD according to the following principle: if the DOD satisfies all local properties, then it also satisfies the global property.

Table 2. *StateOntology*

State Property	Description
is_a_current_requirement(p)	Property *p* is part of the current requirement set.
is_a_subrequirement_of_via(p1, p2, b)	Property *p*1 is a subrequirement of property *p*2 via branch *b*.
undesirable_requirement(p)	Property *p* is an undesirable requirement [e.g., because the stakeholder has indicated that (s)he is not interested in that requirement anymore].
undesirable_branch(b)	Branch *b* is an undesirable branch (e.g., because it contains only undesirable requirements).
undesirable_component(c)	Component *c* is an undesirable component [e.g., because the stakeholder has indicated that (s)he does not like the component].
total_branch_costs(b, x)	The total costs of developing a DOD that satisfies branch *b* are *x*.
intermediate_branch_costs(b, x)	During design, the costs of developing a DOD that satisfies branch *b* are estimated to be *x*.
predicted_costs(b, x)	Before the design process starts, the total costs of developing a DOD that satisfies branch *b* are estimated to be *x*.
requirement_refined(p)	Requirement *p* has currently been refined to subrequirements.
requirement_refined_via(p, b)	Requirement *p* has been refined to subrequirements via branch *b*.
best_branch_for(b, p)	Branch *b* is considered to be the best option to satisfy requirement *p* (e.g., because it has the lowest predicted costs).
best_component_for(c, p)	Component *c* is considered to be the best component to satisfy requirement *p* (e.g., because it has the lowest predicted costs).
costs(c, x)	The costs of developing component *c* are *x*.
'DOD_counter'(x)	As yet, *x* different versions of a DOD have been considered.
current_DOD(DOD(x))	The DOD that is currently considered is called DOD(*x*).
part_of_DOD(c,DOD(x))	Component *c* is part of DOD(*x*).
local_requirement_satisfied(p)	Local requirement *p* is satisfied by the DOD under consideration.
holds_for(p, c)	Property *p* holds for component *c*.

Note: DOD, design object description.

refined to a subrequirement q, and it has not been refined yet, then this should be done by refining via the best branch b (e.g., the one with the lowest predicted costs). Formalization:

is_a_current_requirement(p) ∧ is_a_subrequirement_of_via(q, p, b) ∧ not(requirement_refined(p)) ∧ best_branch_for(b, p) ∧ not(undesirable_branch(b)) ↠ is_a_current_requirement(q) ∧ requirement_refined(p) ∧ requirement_refined_via(p, b)

7.3.2. *Properties concerning the DOD*

The process concerning DODs determines DODs for sets of requirements given as input. Within this process it is taken into account whether or not the stakeholder asserts that certain components are undesirable as part of a design object.

LP6 DOD generation. This property expresses that each local requirement l should be satisfied by adding the best component c for that requirementp to the current DOD, DOD(x). Formalization:

is_a_current_requirement(l) ∧ best_component_for(c, l) ∧ not(undesirable_component(c)) ∧ costs(c, y) ∧ is_a_subrequirement_of_via(l, n, b) ∧ "DOD_counter"(x) ↠ current_DOD(DOD(x)) ∧ part_of_DOD(c, DOD(x)) ∧ intermediate_branch_costs(b, y)

LP8 local requirement satisfaction determination. This property determines when a local requirement l is satisfied by a DOD. This is the case when the current DOD contains a component c for which this requirement holds. Formalization:

current_DOD(d) ∧ part_of_DOD(c, d) ∧ holds_for(l, c) ∧ is_a_current_requirement(l) ↠ local_requirement_satisfied(l)

8. EXAMPLE SIMULATION TRACES

Using the simulation model described in Section 7, a number of experiments were performed. In such experiments, different types of revision might be needed with an increasing impact on the design process:

- revision of the *DOD* for given requirements based on the stakeholders judgment that a component used in the DOD is undesirable,
- revision of the refined *requirements* based on the stakeholder's judgment that one of these requirements is undesirable, and
- revision of a whole *branch* based on the calculation that the costs of the DOD found are higher than expected.

The simulation has been performed within the LEADSTO software environment. Basically, the temporal rules are processed (in parallel) over time, thereby maintaining time intervals for which certain state properties hold. For an extensive description of the algorithm, see Bosse et al. (2007).

The first trace depicted in Figure 3 shows a design process in which no revision is needed. In this and the next figure, time is on the horizontal axis, the derived state properties

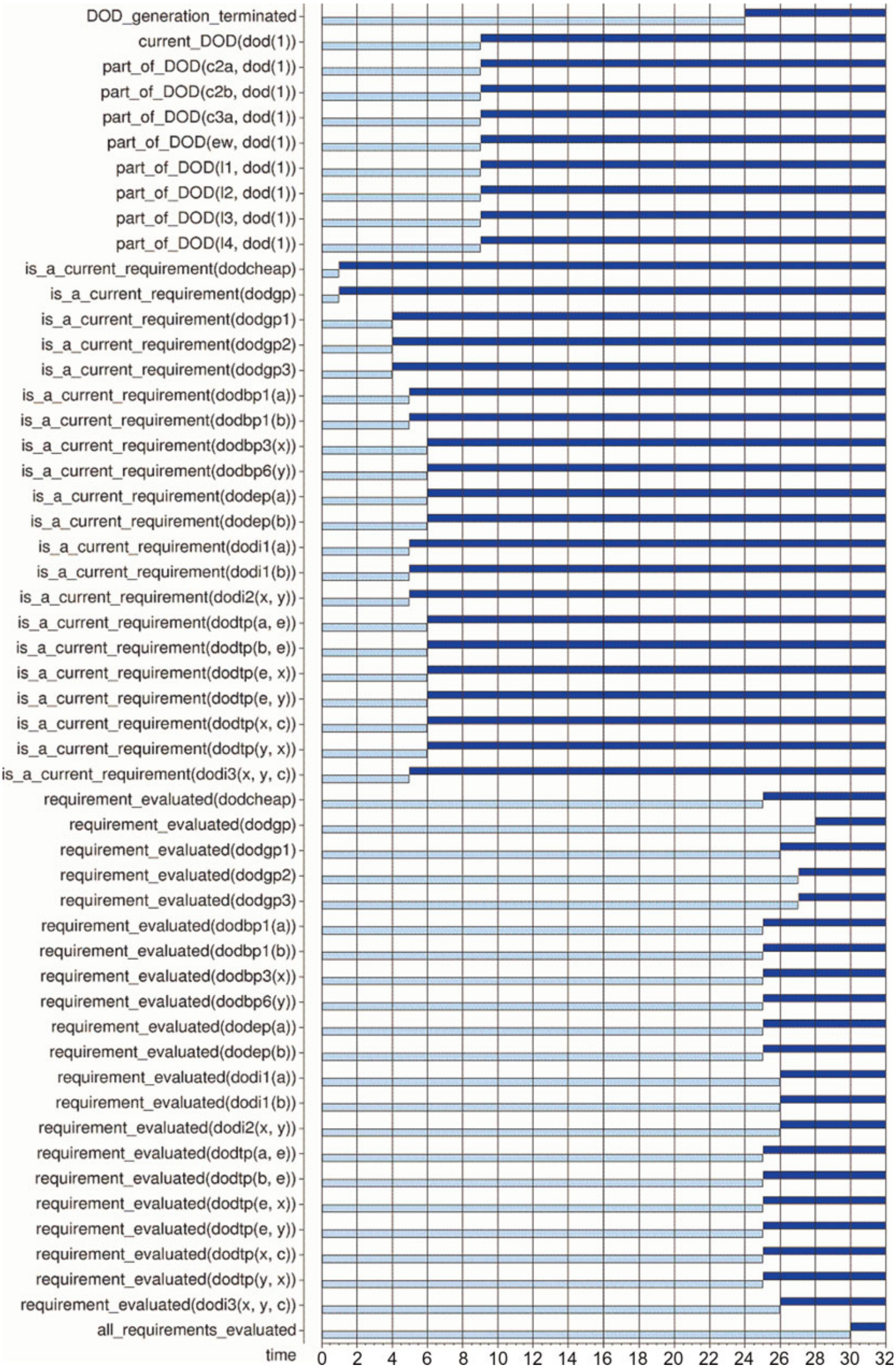

Fig. 3. Simulation trace 1. [A color version of this figure can be viewed online at journals.cambridge.org/aie]

are on the vertical axis. A dark box on top of a line indicates that a state property is true at that time point, whereas a light box below a line indicates that a state property is false. To facilitate understanding, only the most relevant of the derived atoms are shown (not in any particular order). In the simulations shown in this paper, for all local properties the values (0, 0, 1, 1) have been chosen for the timing parameters e, f, g, and h. This means that for each local property, the consequent is derived for one time unit directly (with zero delay) after the antecedent has been true for one time unit. In cases the modeler desires to specify the durations of the different parts of the design process explicitly, these parameters can be replaced by choosing more realistic values (e.g., in terms of seconds) for each individual local property.

As shown in Figure 3, when the process starts, first the initial requirements DODGP and DODCHEAP are identified. After this, these requirements are refined into subrequirements DODGP1, DODGP2, and DODGP3 (based on the logical relationships of the tree in Fig. 2, also see the representation of this tree by means of the relation is_a_subrequirement_of_via in Section 7). This process continues until the most elementary requirements (i.e., those that have no subrequirements; the leaves of the tree) have been reached. Then a new DOD [called DOD(1)] is created, which consists of a number of components (and connections between them) that satisfy all local requirements.

During this process, if possible only the components with the lowest predicted costs are selected, according to a simple mechanism that prefers branches with lower costs (see local property LP3 for the implementation). As soon as a satisfactory DOD has been found (at least according to the requirements that were derived), DOD generation finishes, and after this, all (local and nonlocal) requirements that are part of the DOD are evaluated once more (this time based on the logical relationships represented by the relation is_implied_by(. . .) above). As they all turn out to be satisfied (see the requirement_evaluated(. . .) atoms), the design process terminates.

An example of a process where revision is needed is shown in Figure 4. Initially, this trace has exactly the same dynamics as the previous one. In the beginning, only the requirements DODGP and DODCHEAP are present. Then, requirements are refined until the leaves of the tree (the local requirements) have been reached, and subsequently a DOD is selected using the same components as in trace 1, but at time point 9 something different happens. Here, the atom undesirable_component(C2A) becomes true (representing the fact that the stakeholder has indicated that this component is not desirable). As a consequence, that component is removed from the current DOD and replaces it by another (probably more expensive) component, C1A. Finally, the design process succeeds in finding a satisfactory DOD. This resulting DOD is then evaluated and its total costs are calculated.

In addition to the simulation trace shown here, two other examples of simulation traces are described in Appendix C. Because these kinds of simulations can be performed at an abstract level, they are very appropriate to perform (pseudo)experiments. Such experiments may be useful, for example, for analysts of design processes, because it allows them to explore what happens in certain critical situations, without having to perform actual design processes.

9. GLOBAL DYNAMIC PROPERTIES OF A DESIGN PROCESS

For design processes like the one described above, in addition to local dynamic properties, a number of global dynamic properties can be identified that are expected to hold.

- During (or after termination of) the design process, the design process objectives are fulfilled. After termination of the design process the final DOD satisfies the requirements of the final requirements state.
- After termination of the design process the requirements in the final requirements state have been declared sufficient by the stakeholder at some point during the process.
- If one of the design process objectives is that the design process should be fast and cheap, then any DOD generated during the process solely consists of standard components.

Section 9.1 presents a number of such dynamic properties expressed as TTL statements. These properties as listed are relevant to be considered and were checked for a number of design reasoning traces. They need not be satisfied by all design reasoning traces; they may be used to distinguish between different types of design reasoning traces as well. Next, Section 9.2 introduces the TTL Checking Tool, which can be used to verify such properties against (simulated and empirical) traces. Section 9.3 presents the results of checking the properties presented in Section 9.1 against a number of simulated traces. Finally, in Section 9.4 the advantages of this approach are discussed, in particular, in comparison with verification approaches such as model checking.

9.1. Global dynamic properties

The following global properties have been identified and formally specified in TTL.

GP1 local requirement satisfaction: Eventually there is a DOD that contains a satisfactory component for each local requirement that exists at that moment. Formalization:

$$\begin{aligned}&\exists t\ \exists d{:}DOD\ \mathrm{state}(\gamma, t) \models \mathrm{current\_DOD}(d)\ \&\ \forall r{:}\mathrm{localreq}\\&\quad[\mathrm{state}(\gamma, t) \models \mathrm{is\_a\_current\_requirement}(r)\\&\quad\Rightarrow \exists\ c{:}\mathrm{component}\ \mathrm{state}(\gamma, t) \models \mathrm{part\_of\_DOD}(c, d)\\&\quad\&\ \mathrm{state}(\gamma, t) \models \mathrm{holds\_for}(r, c)]\end{aligned}$$

GP2 termination of the design process: Eventually the process will terminate. Formalization:

$$\exists t\ \mathrm{state}(\gamma, t) \models \mathrm{DOD\_generation\_terminated}$$

GP3 cheapest components per local requirement: For each local requirement, if there is a component that satisfies it, then

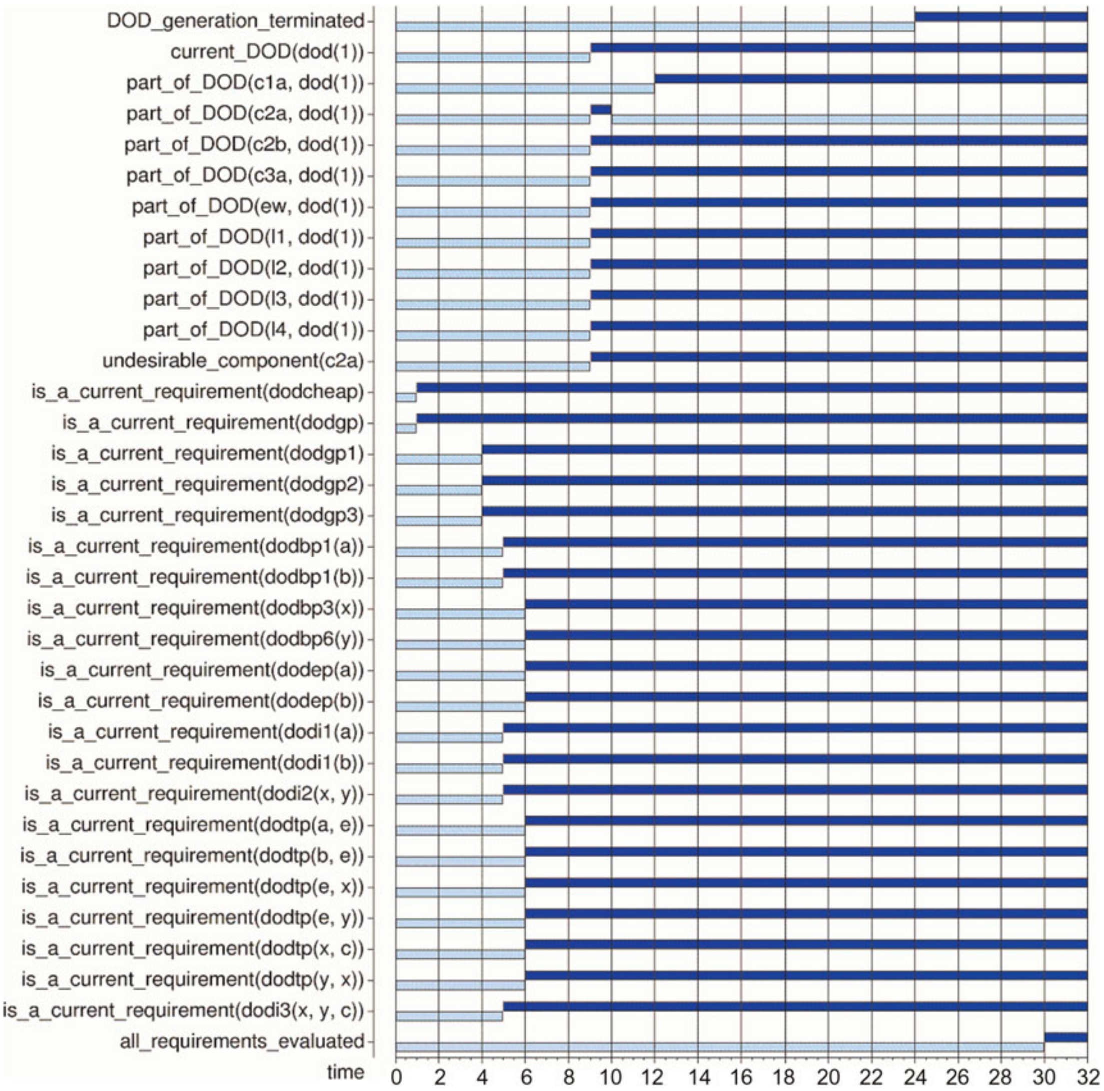

Fig. 4. Simulation trace 2. [A color version of this figure can be viewed online at journals.cambridge.org/aie]

the cheapest component that satisfies it will be added to the DOD. (Note that this is a refinement of GP4.) Formalization:

$$\begin{aligned}&\forall t\ \forall r{:}\text{localreq}\ \forall c{:}\text{component state}(\gamma, t) \models \text{is\_a\_current\_requirement}(r)\ \&\ \text{state}(\gamma, t) \models \text{holds\_for}(r, c)\\&\quad \Rightarrow \exists t' \geq t\ \exists d{:}\text{DOD}\ \exists c'{:}\text{component}\ \exists p{:}\text{integer state}(\gamma, t')\\&\quad \models \text{part\_of\_DOD}(c', d)\ \&\ \text{state}(\gamma, t') \models \text{holds\_for}(r, c')\\&\quad \&\ \text{state}(\gamma, t') \models \text{costs}(c', p)\ \&\ \neg\ [\exists c''{:}\text{component}\ \exists p' < p\\&\quad \text{state}(\gamma, t') \models \text{holds\_for}(r, c'')\ \&\ \text{state}(\gamma, t') \models \text{costs}(c'', p')]\end{aligned}$$

GP4 DOD successfulness: For each local requirement, if there is a component that satisfies it, then such a component will be added to the DOD. Formalization:

$$\begin{aligned}&\forall t\ \forall r\ \forall{:}\text{localreq}\ \forall c{:}\text{component state}(\gamma, t) \models \text{is\_a\_current\_requirement}(r)\ \&\ \text{state}(\gamma, t) \models \text{holds\_for}(r, c)\\&\quad \Rightarrow \exists t' \geq t\ \exists d{:}\text{DOD}\ \exists c'{:}\text{component state}(\gamma, t') \models \text{part\_of\_DOD}(c', d)\ \&\ \text{state}(\gamma, t') \models \text{holds\_for}(r, c')\end{aligned}$$

GP5(c) total costs: Eventually the system generates a DOD of which the costs are exactly *c*. Formalization:

$$\exists t\ \text{state}(\gamma, t) \models \text{total\_DOD\_costs}(c)$$

GP6 requirement persistence: Once it is derived that a requirement is needed for the system, this requirement persists forever. Formalization:

$$\begin{aligned}&\forall t\ \forall r{:}\text{req state}(\gamma, t) \models \text{is\_a\_current\_requirement}(r)\\&\quad \Rightarrow \forall t' \geq t\ \text{state}(\gamma, t') \models \text{is\_a\_current\_requirement}(r)\end{aligned}$$

GP7 new DOD grounding: If an old DOD is replaced by a new one, then there is an undesirable branch. Formalization:

$$\begin{aligned}&\forall t \forall t' > t\ \forall x{:}\text{integer}\ [\text{state}(\gamma, t) \models \text{current\_DOD}(x)\\&\quad \&\ \text{state}(\gamma, t') \models \text{current\_DOD}(x + 1)\ ]\\&\quad \Rightarrow \exists b{:}\text{branch state}(\gamma, t') \models \text{undesirable\_branch}(b)\end{aligned}$$

GP8 requirements refinement successfulness: At a certain point in time, all nonlocal requirements will be refined. Formalization:

$$\exists t\ \forall n{:}\mathrm{nonlocalreq}\ \mathrm{state}(\gamma, t) \models \mathrm{is\_a\_current\_requirement}(n) \Rightarrow \mathrm{state}(\gamma, t) \models \mathrm{requirement\_refined}(n)$$

GP9 cheap requirement satisfaction: If there is a requirement that the system should be cheap, then eventually a DOD will be of which the costs are at most δ. In the current case study, we use $\delta = 1500$. Formalization:

$$\forall t\ \mathrm{state}(\gamma, t) \models \mathrm{is\_a\_current\_requirement}(\mathrm{DODCHEAP}) \Rightarrow \exists t2 > t\ \exists x{:}\mathrm{integer}\ \mathrm{state}(\gamma, t) \models \mathrm{total\_DOD\_costs}(x)\ \&\ x \leq \delta$$

Note that all global properties shown in this section, as well as the local properties introduced in Section 7.3 are generic. As can be seen from their formalizations, they contain (domain-independent) variables for requirements, components, DODs, and so on. As a result, it is relatively easy to reuse them for any arbitrary design process. Nevertheless, in case a different domain is chosen, obviously the domain specific information represented as facts (see Section 7.2) should be filled in for the new domain.

9.2. The TTL Checking Tool

To enable automated verification of TTL properties against traces, a dedicated software tool has been developed: the TTL Checking Tool (Bosse et al., 2006*a*). This tool, which was implemented in SWI-Prolog, takes a formal TTL property and a set of traces as input, and determines whether the property holds for the traces or not. In case the property does not hold, the software provides a counter example, that is, a particular combination of instantiated variables for which the property fails. Traces may be obtained in several ways, for example, by simulation, but also based on empirical data.

The verification algorithm is a backtracking algorithm that systematically considers all possible instantiations of variables in the TTL formula under verification. However, not for all quantified variables in the formula the same backtracking procedure is used. Backtracking over variables occurring in "holds atoms" [e.g., the variable *n* in $\mathrm{state}(\gamma, t) \models \mathrm{is\_a\_current\_requirement}(n)$] is replaced by backtracking over values occurring in the corresponding holds atoms in traces under consideration. Because there are a finite number of such state atoms in the traces, iterating over them often will be more efficient than iterating over the whole range of the variables occurring in the holds atoms. Formulae that contain variables quantified over infinite sorts not occurring in a holds atom cannot be checked by the TTL checker.

As time plays an important role in TTL formulae, special attention is given to continuous and discrete time range variables. Because of the *finite variability* property of TTL traces (i.e., only a finite number of state changes occur between any two time points), it is possible to partition the time range into a minimum set of intervals within which all atoms occurring in the property are constant in all traces. Quantification over continuous or discrete time variables is replaced by quantification over this finite set of time intervals.

To increase the efficiency of verification, the TTL formula that needs to be checked is compiled into a Prolog clause. Compilation is obtained by mapping conjunctions, disjunctions and negations of TTL formulae to their Prolog equivalents, and by transforming universal quantification into existential quantification. Thereafter, if this Prolog clause succeeds, the corresponding TTL formula holds with respect to all traces under consideration.

The complexity of the algorithm has an upper bound in the order of the product of the sizes of the ranges of all quantified variables. However, if a variable occurs in a holds atom, the contribution of that variable is no longer its range size, but the number of times that the holds atom pattern occurs (with different instantiations) in trace(s) under consideration. The contribution of an isolated time variable is the number of time intervals into which the traces under consideration are divided.

The specific optimizations discussed above make it possible to check realistic dynamic properties with reasonable performance. To give an impression, all of the checks mentioned in the next section can be performed within a couple of seconds. With the increase of the number of traces with similar complexity as the first one, the verification time grows linearly. However, the verification time is polynomial in the number of isolated time range variables occurring in the formula under verification. Nevertheless, the complexity of the checking process is still much smaller than that of exhaustive forms of verification such as model checking (see Section 9.4 for an elaborate discussion).

9.3. Checking results

In this section, the properties mentioned in Section 9.1 have been checked against four different simulation traces (i.e., those shown in Section 8 and Appendix C). As can be seen in Table 3, most global properties hold for all traces. For trace 2, GP3 does not hold. The reason for this is that component c2a (which is the cheapest component for several requirements) is eventually rejected by the stakeholder. Property GP6 does not hold for traces 3 and 4, because in these traces requirement revision takes place. Obviously, property GP5(1403) only holds for trace 4, because in the other traces the final DODs have different costs.

9.4. Checking of traces

Within the literature on analysis of properties (verification), generally two types of verification are distinguished: an empirical type of verification or validation [i.e., verifying whether a certain property holds for a (limited) set of traces],

Table 3. *Results of automated checking*

Property	Trace 1	Trace 2	Trace 3	Trace 4
GP1	+	+	+	+
GP2	+	+	+	+
GP3	+	−	+	+
GP4	+	+	+	+
GP5(1403)	−	−	−	−
GP6	+	+	−	−
GP7	+	+	+	+
GP8	+	+	+	+
GP9	+	+	+	+

and an exhaustive type of verification [i.e., verifying whether a certain property holds for a model (thus for all possible traces) of a system]. In the literature, much emphasis is put on the latter type of analysis, of which model checking (McMillan, 1993; Clarke et al., 1999) is one of the most famous examples. This essentially comes down to the problem of justifying entailment relations between sets of properties defined at different aggregation levels of a system's representation. In general, entailment relations can be established either by logical proof procedures or by checking properties of a higher aggregation level on the set of all theoretically possible traces generated by executing a system specification that consists of properties of a lower aggregation level. To make that feasible, expressivity of the language for these properties has to be sacrificed to a large extent (because for many relevant properties of design processes, an exhaustive verification is simply undecidable). However, checking properties on a practically given set of traces (instead of all theoretically possible ones) is computationally much cheaper, and therefore, the language for these properties can be more expressive (see Clarke et al., 1999; Bosse et al., 2006*a*) for an extensive discussion about this topic). TTL is an example of such an expressive language. For example, the possibility of explicit reference to time points and time durations enables modeling of the dynamics of continuous real-time phenomena. This feature goes beyond the expressive power available in standard linear or branching time temporal (modal) logics such as LTL and CTL (e.g., van Benthem, 1983; Goldblatt, 1992). Furthermore, the possibility to quantify over traces in TTL allows for specification of more complex adaptive behaviors, such as the property "exercise improves skill." This is a relative property in the sense that it involves the comparison of two alternatives for the history. Similarly, in the context of the current paper, an example of such a relative property would be "for any two traces $\gamma 1$ and $\gamma 2$, if in $\gamma 1$ there are more initial requirements than in $\gamma 2$, then the design process takes longer in $\gamma 1$." This is a property that does not necessarily hold for all traces, but it can be useful in order to distinguish different classes of design processes. These kinds of relative properties can easily be expressed in TTL, whereas in standard forms of temporal logic different alternative histories cannot be compared. Moreover, the presented framework allows the designer to generate a large number of simulation traces, and automatically check in which situations the properties hold and when they do not hold.

10. LOGICAL INTERLEVEL RELATIONSHIPS BETWEEN DYNAMIC PROPERTIES

In addition to the above, logical relationships can be and have been identified between dynamic properties at different abstraction levels. Such *interlevel relations* relate the global properties presented in this section to some of the local properties presented in Section 7. They can be specified by means of logical implications or graphically by means of AND/OR trees (see also Jonker & Treur, 2002). In these relationships, also properties at an intermediate level of aggregation (intermediate properties) occur, addressing smaller steps than global properties do, but bigger steps than local properties do. Such interlevel relations can play an important role in the analysis of design processes, because of their hierarchical structure. In case the properties involved are of reasonable complexity, the interlevel relations can be automatically verified using techniques from (McMillan, 1993; Clarke et al., 1999; Sharpanskykh & Treur, 2006). However, as mentioned in the previous section, this is often not feasible. In such situations, the approach of checking the global properties against simulation traces that were generated on the basis of the local properties might be a useful alternative. Nevertheless, in this section, as an example some relationships between global properties and local properties of a design process are discussed. For the purposes of presentation they will only be described in an informal/semiformal form. The exhaustive verification of the relationships (using model checking or similar techniques) is, however, beyond the scope of this article.

One of the most relevant global properties of a design process is whether or not a set of requirements and a DOD are generated such that the DOD fulfills the set of requirements and both satisfy the stakeholder. However, without further assumptions this property is not guaranteed. Some of the reasons why it may be hard to come to a result are the following:

- the stakeholder may impose a set of requirements that is too strong (inconsistent), such as wanting a very cheap solution of very high quality, and is not willing to compromise them;
- the stakeholder may keep changing his or her mind on whether certain requirements or design object components are undesirable; and
- the available set of components is too limited to fulfill the stakeholder's requirements.

In the first case the requirements can be inconsistent so that no design object exists that fulfills them all. In that case the outcome of such a design process asserts this. In the second case a design process may go on forever, all the time adapting to the latest preferences of the stakeholder, but never coming to an end: every DOD or requirement set is rejected by the stakeholder. The third case may have a similar outcome as the first case.

Under reasonable environment assumptions on the stakeholder's behavior, however, it may be guaranteed that a design process has a successful outcome. Especially if finiteness assumptions are made for the number of different types of components for DODs that are available, and for the sets of requirements that are possible, such assumptions may be reasonable. To exclude the cases mentioned above, environment assumptions made as described below. Here, the following abbreviation is used for a state property p to express that in trace γ this state properties stabilizes:

$$\begin{aligned}
&\text{stabilizes}(\gamma, \text{p}, \text{pos}) \equiv \exists \text{t} \forall \text{t}' \geq \text{t state}(\gamma, \text{t}') \models \text{p}\\
&\text{stabilizes}(\gamma, \text{p}, \text{neg}) \equiv \exists \text{t} \forall \text{t}' \geq \text{t state}(\gamma, \text{t}') \models \text{not(p)}\\
&\text{stabilizes}(\gamma, \text{p}) \equiv \exists \text{t } [\forall \text{t}' \geq \text{t state}(\gamma, \text{t}') \models \text{p} \vee \forall \text{t}' \geq \text{t}\\
&\quad \text{state}(\gamma, \text{t}') \models \text{not(p)}]
\end{aligned}$$

10.1. EP1

After some time the stakeholder's (un)desirability preferences for requirements stabilize: a time point exists after which he or she does not provide any new input with respect to (un)desirability of requirements. Formally:

$$\forall \gamma\ \forall \text{r stabilizes}(\gamma, \text{undesirable\_requirement(r)})$$

10.2. EP2

After some time the stakeholder's (un)desirability preferences for DOD components stabilize: a time point exists after which he or she does not provide any new input with respect to (un)desirability of components. Formally,

$$\forall \gamma\ \forall \text{c stabilizes}(\gamma, \text{undesirable\_component(c)})$$

10.3. EP3

At least one fully refined set of requirements exists that does not contradict a stakeholder's stabilized (un)desirability preferences for requirements. Formally,

$$\begin{aligned}
&\exists \gamma\ \exists \text{r}_1, \ldots \text{r}_\text{n} \text{ } [\ \forall \text{r } [\text{ stabilizes}(\gamma, \text{R(r), pos}) \Leftrightarrow \vee_\text{i} \text{ r} = \text{r}_\text{i} \text{ }]\\
&\quad \&\ \forall \text{r [stabilizes}(\gamma, \text{undesirable\_requirement(r), pos})\\
&\quad \Rightarrow \text{not stabilizes}(\gamma, \text{R(r), pos)] } \& \forall \text{r [stabilizes}(\gamma, \text{R(r),}\\
&\quad \text{pos}) \Rightarrow [\text{ stabilizes}(\gamma, \text{undesirable\_requirement(r), neg})\\
&\quad \vee [\text{ stabilizes}(\gamma, \exists \text{ r}', \text{b is\_a\_subrequirement\_of\_via(r, r}', \text{b})\\
&\quad \wedge \text{R(r}'\text{), pos) \& stabilizes}(\gamma, \text{R(r}'\text{), pos)]] } \& \forall \text{r stabilizes}\\
&\quad (\gamma, \text{R(r), pos}) \Rightarrow [\ [\text{ stabilizes}(\gamma, \exists \text{ q, b is\_a\_subrequirement\_}\\
&\quad \text{of\_via(q, r, b)} \wedge \text{R(q), pos) \& stabilizes}(\gamma, \text{R(q), pos})]\\
&\quad \vee \exists \text{c stabilizes}(\gamma, \text{holds\_for(r, c), pos)]]}
\end{aligned}$$

10.4. EP4

At least one DOD exists that fulfills a set of fully refined requirements that does not contradict a stakeholder's stabilized (un)desirability preferences for requirements, and that does not contradict the stakeholder's stabilized (un)desirability preferences for components. Formally,

$$\begin{aligned}
&\exists \gamma\ \exists \text{r}_1, \ldots \text{r}_\text{n} \text{ } [\ \forall \text{r } [\text{ stabilizes}(\gamma, \text{R(r), pos}) \Leftrightarrow \vee_\text{i} \text{ r} = \text{r}_\text{i} \text{ }]\\
&\quad \&\ \forall \text{r [stabilizes}(\gamma, \text{undesirable\_requirement(r), pos})\\
&\quad \Rightarrow \text{not stabilizes}(\gamma, \text{R(r), pos)] } \&\ \forall \text{r [stabilizes}(\gamma, \text{R(r), pos})\\
&\quad \Rightarrow [\text{ stabilizes}(\gamma, \text{undesirable\_requirement(r), neg})\\
&\quad \vee [\text{ stabilizes}(\gamma, \exists \text{ r}', \text{b is\_a\_subrequirement\_of\_via(r, r}', \text{b})\\
&\quad \wedge \text{R(r}'\text{), pos) \& stabilizes}(\gamma, \text{R(r}'\text{), pos)]] } \& \forall \text{r stabilizes}\\
&\quad (\gamma, \text{R(r), pos}) \Rightarrow [\ [\text{ stabilizes}(\gamma, \exists \text{ q, b is\_a\_subrequirement\_}\\
&\quad \text{of\_via(q, r, b)} \wedge \text{R(q), pos) \& stabilizes}(\gamma, \text{R(q), pos})]\\
&\quad \vee \exists \text{c stabilizes}(\gamma, \text{holds\_for(r, c), pos)] } \& \exists \text{d stabilizes}\\
&\quad (\gamma, \forall \text{r R(r)} \Rightarrow \exists \text{c part\_of (c, d) \& holds\_for(r, c), pos)]}
\end{aligned}$$

Note that, also under such assumptions (leaving no doubt on the existence of a suitable requirement set and DOD), the design process has to face serious challenges: for example, the challenge to uncover such a stable requirements set and the challenge to find such a DOD. Moreover, note that the assumptions do not imply a unique stable stakeholder situation: different design traces may exist in which the stakeholder's stable preferences are different.

Next, the relevant dynamic properties of different parts of the design process are considered. For the construction and maintenance of the DOD two properties are relevant. The first expresses that if after some time the stakeholder does not change his or her mind about (un)desirability of components, then after some time the branch costs stabilize. This property is related to the fact that the number of branches is finite, and that for a given (stabilized) set of undesirable components, for each branch a unique number is determined. These properties can be formalized in a manner similar as the ones above.

10.5. IPDOD1

IF: after some time the stakeholder's (un)desirability preferences for design object description components stabilize

AND: after some time a stabilized fully refined stable set of current requirements occurs

THEN: after some time the branch costs stabilize

The second property expresses that if for the given context a design solution exists, it will be generated.

10.6. IPDOD2

IF: after some time a stabilized fully refined stable set of current requirements occurs

AND: after some time the stakeholder's (un)desirability preferences for design object description components stabilize

AND: at least one DOD exists that fulfills the stabilized set of fully refined requirements that does not contradict the stakeholder's stabilized (un)desirability preferences for components, with branch costs below the expected branch costs

THEN: after some time a stabilized DOD occurs that fulfills the stabilized set of fully refined requirements that does not contradict the stakeholder's stabilized (un)-desirability preferences for components, with branch costs below the expected branch costs

For the processes related to requirements it is expressed if the input received satisfies the environmental assumptions given above, then a stable set of current requirements will occur.

10.7. IPRQ1

IF: after some time the stakeholder's (un)desirability preferences for requirements stabilize

AND: after some time the stakeholder's (un)desirability preferences for design object description components stabilize

AND: after some time the branch costs stabilize

AND: at least one DOD exists that fulfills a set of fully refined requirements that does not contradict a stakeholder's stabilized (un)desirability preferences for requirements and that does not contradict the stakeholder's stabilized (un)desirability preferences for components

THEN: after some time a stabilized fully refined stable set of current requirements occurs that does not contradict the stakeholder's stabilized (un)desirability preferences for requirements

The top level property that was considered is GP0.

10.8. GP0

After some time a stabilized fully refined stable set of current requirements occurs that does not contradict a stakeholder's stabilized (un)desirability preferences for requirements; and after some time a stabilized DOD occurs that fulfills this stabilized set of fully refined requirements that does not contradict the stakeholder's stabilized (un)desirability preferences for components, with branch costs below the expected branch costs.

The logical interlevel relationships are as follows:

$$\text{EP1 \& EP2 \& EP3 \& EP4 \& IPDOD1 \& IPDOD2 \& IPRQ1} \Rightarrow \text{GP0}$$

where EP represents all (required) environmental properties. These interlevel relationships are depicted in the graphical form of an AND-tree in Figure 5. Notice the difference between the trees depicted in Figure 2 and in Figure 5. The former tree is about properties of the *design object*, whereas the latter shows properties of the *design process*.

In combination with the automated checks of simulation traces described in Section 9, a hierarchy of interlevel relations can play an important role in the analysis of design processes. More specifically, if a certain global property turns out not to hold for a given design process trace (either a simulation trace or an empirical trace), then in a top-down fashion, the logical relationships can be consulted in order to pinpoint which local properties are candidates for causing the failure, and hence, to be verified in a given trace of a design process. For example, suppose for a given trace it has been detected (using the checker tool described in Section 9.2) that the dynamic property GP0 at the highest aggregation level of the tree does not hold, that is, no satisfactory set of requirements or DOD is generated. Then, given the AND-tree structure in Figure 5, at least one of the children of GP0 will not hold (if they all would hold for the given trace, also GP0 would hold for this trace), which means that EP, IPRQ1, IPDOD1, or IPDOD2 will not hold. Suppose by further checking it is found that EP does not hold. Then the diagnostic process can be continued by focusing on this property. It follows that EP1, EP2, EP3, or EP4 does not hold (see Fig. 5). Checking these four properties will pinpoint the cause of failure. Notice that this diagnostic process is economic in the sense that, if EP holds, the whole subtree under EP is not examined, because there is no reason for that.

11. DISCUSSION

To develop automated support for the dynamics of nontrivial design processes, the challenge of modeling and analyzing such dynamics in a formal manner has to be addressed (cf.

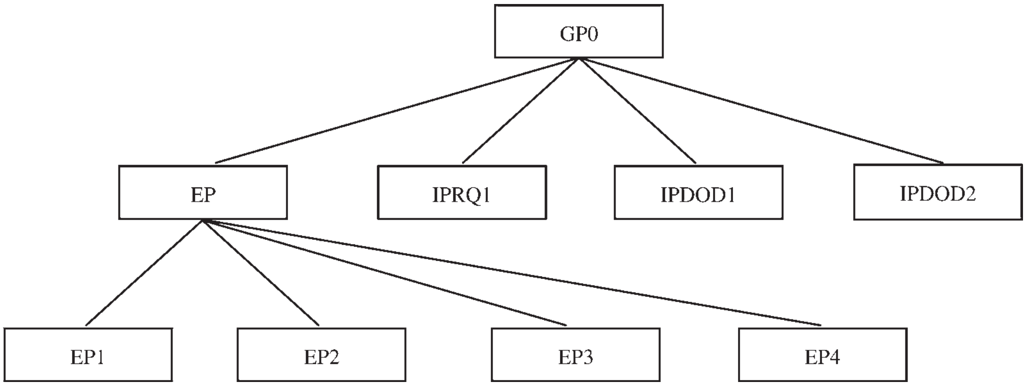

Fig. 5. The logical interlevel relationships between the dynamic properties of a design process.

Brown & Chandrasekaran, 1989; Baldwin & Chung, 1995; Corkill, 2000; Heller & Westfechtel, 2003). In the literature about the application of formal techniques to design processes, often a distinction is made between simulation methods and verification methods (see, e.g., Kern & Greenstreet, 1999). Basically, simulation methods are used to provide the user more insight in the dynamics of a particular design process. For example, Mavris et al. (1999) apply probabilistic simulation techniques to study the dynamics of design processes over time and meanwhile deal with aspects of uncertainty. Similarly, Sosa and Gero (2005) describe (creative) design as a social process, of which they study the dynamics using multiagent-based simulation. Conversely, verification methods can be applied to prove certain expected properties of design processes, such as successfulness and efficiency. Among the approaches used are model checking (McMillan, 1993; Clarke et al., 1999), but also automata-theoretic techniques (Thomas, 1990) and theorem proving techniques (Duffy, 1991). This paper proposes some first steps to integrate (a restricted variant of) verification techniques (the idea of checking properties against a set of traces) with simulation techniques. More specifically, the complex dynamics of a design process has been analyzed in such a precise way that properties of the process as a whole can be specified and, moreover, part of the analysis contains enough detail to allow for simulation. The results of the simulation have been checked against the properties of the design process as a whole. As has been shown, this type of analysis is computationally cheaper than exhaustive verification approaches such as model checking (McMillan, 1993; Clarke et al., 1999), which allows the properties to be checked to be more expressive. Obviously, an inevitable drawback is the fact that there is no guarantee that the selected traces cover all possible scenarios.

Compared to the references mentioned above, the approach put forward is a declarative, hybrid logical and numerical approach supported by a formal language TTL for specification of dynamic properties of design processes, which has a high expressivity, including the use of variables over sorts, and sorts for real and integer numbers, and arithmetical calculations for these sorts (cf. Jonker & Treur, 2002; Bosse et al., 2006*a*). Because of this expressivity, the full TTL is undecidable and full model checking will not work in general. However, checking properties on given traces can be done in an efficient manner.

Furthermore, also simulation models are specified in a declarative, both logical and numerical manner, in the language LEADSTO, which allows using these specifications in a declarative analysis as well (see also Bosse et al., 2008). Both TTL and LEADSTO assume (simulated) parallel processing, expressed by states at each point in time that include all state properties, and in LEADSTO a processing algorithm for the temporal rules that execute them in parallel.

An example of the use of TTL is establishing logical interlevel relationships between dynamic properties of different levels (cf. Bosse et al., 2007). Alternative approaches such as CSP, temporal logic, interval temporal logic, and assumption–commitment specifications, do not allow such an expressive hybrid representation format.

The paper shows the potential of this formal analysis as a technique for analysis at a high level of abstraction, and for constructing simulations at an abstract level to experiment with dynamics of a design process. The simulation actually is entailed by the analysis and requires no additional programming, thus basically, getting a simulation for free when doing an analysis. Such experiments may be particularly beneficial for analysts of design processes, since it allows them to perform "what-if" analyses, that is, to explore what happens in certain critical situations, without having to perform actual design processes.

Furthermore, the presented approach is generic in the sense that the presented properties of design processes as well as the analysis techniques are independent of any specific design problem. The analysis approach that is for the first time applied to design processes here, has (at least in part) previously been applied to complex and dynamic reasoning processes other than design, such as reasoning by dynamically adding and evaluating assumptions (Jonker & Treur, 2003; Bosse et al., 2006*b*), and reasoning based on multiple representations (Bosse et al., 2003). In these cases in addition to simulated traces, empirical (human) reasoning traces have also been formally analyzed. For further research it is planned to formally analyze protocols of human design processes in a similar manner, using methods as, for example, described in Nagai and Taura (2006).

To have a solid foundation for process descriptions is considered a fundamental issue for design science, which needs to be considered (e.g., Smithers, 1996, 1998). This is a main focus of this paper. Another central foundational issue that has a close relationship to the work reported here is the challenge to develop a general theory of design. In the past, a few authors have addressed this challenge and contributed some proposals or discussions (see, e.g., Yoshikawa, 1981; Tomiyama & Yoshikawa, 1985; Dixon, 1987; Hubka & Eder, 1988, 1995; Treur, 1991; Tomiyama, 1994; Warfield, 1994; Reich, 1995; Brazier et al., 1996; Smithers, 1996, 1998; Hooker, 2004). The foundational approach contributed in this paper may provide new input for further developments on the area of design theory.

ACKNOWLEDGMENTS

An earlier, shorter report of part of this work was presented at ECAI 2004. The paper benefited from the constructive comments by the anonymous reviewers.

REFERENCES

Baldwin, R.A., & Chung, M.J. (1995, February). A formal approach to managing design processes. *IEEE Computer* 54–63.

Barringer, H., Fisher, M., Gabbay, D., Owens, R., & Reynolds, M. (1996). *The Imperative Future: Principles of Executable Temporal Logic*. New York: Wiley.

Bosse, T., Jonker, C.M., & Treur, J. (2003). Simulation and analysis of controlled multi-representational reasoning processes. *Proc. 5th Int. Conf. Cognitive Modelling, ICCM'03*, pp. 27–32. Universitats-Verlag Bamberg.

Bosse, T., Jonker, C.M., & Treur, J. (2006). Reasoning by assumption: formalisation and analysis of human reasoning traces. *Cognitive Science Journal 20*, 147–180.

Bosse, T., Jonker, C.M., van der Meij, L., Sharpanskykh, A., & Treur, J. (2006). Specification and verification of dynamics in cognitive agent models. *Proc. 6th Int. Conf. Intelligent Agent Technology, IAT'06* (Nishida, T., Klusch, M., Sycara, K., & Yokoo, M., Eds.), pp. 247–254. New York: IEEE Computer Society Press.

Bosse, T., Jonker, C.M., van der Meij, L., & Treur, J. (2007). A language and environment for analysis of dynamics by SimulaTiOn. *International Journal of Artificial Intelligence Tools 16(3)*, 435–464.

Bosse, T., Sharpanskykh, A., & Treur, J. (2008). Modelling complex systems by integration of agent-based and dynamical systems models. *Proc. 6th Int. Conf. Complex Systems, ICCS'06* (Minai, A., Braha, D., & Bar-Yam, Y., Eds.). New York: Springer–Verlag.

Brazier, F.M.T., van Langen, P.H.G., Ruttkay, Zs., & Treur, J. (1994). On formal specification of design tasks. *Artificial Intelligence in Design '94, Proc. AID'94* (Gero, J.S., & Sudweeks, F., Eds.), pp. 535–552. Dordrecht: Kluwer Academic.

Brazier, F.M.T., van Langen, P.H.G., & Treur J. (1996). A logical theory of design. *Advances in Formal Design Methods for CAD, Proc. 2nd Int. Workshop Formal Methods in Design* (Gero, J.S., Ed.), pp. 243–266. New York: Chapman & Hall.

Brown, D.C., & Chandrasekaran, B. (1989). *Design Problem Solving: Knowledge Structures and Control Strategies*. London: Pitman.

Clarke, E.M., Grumberg, O., & Peled, D.A. (1999). *Model Checking*. Cambridge, MA: MIT Press.

Corkill, D.D. (2000). When Workflow doesn't work: issues in managing dynamic processes, *Proc. Design Project Support using Process Models Workshop, 6th Int. Conf. Artificial Intelligence in Design*, pp. 1–13.

Dixon, J.R. (1987). On research methodology towards a scientific theory of engineering design. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing 1*, 145–157.

Duffy, D.A. (1991). *Principles of Automated Theorem Proving*. New York: Wiley.

Fisher, M. (2005). Temporal development methods for agent-based systems. *Journal of Autonomous Agents and Multi-Agent Systems 10*, 41–66.

Galton, A. (2003). Temporal logic. *Stanford Encyclopedia of Philosophy*. Accessed at http://plato.stanford.edu/entries/logic-temporal/#2

Galton, A. (2006). Operators vs arguments: the ins and outs of reification. *Synthese 150*, 415–441.

Gavrila, I.S., & Treur, J. (1994). A formal model for the dynamics of compositional reasoning systems. *Proc. 11th European Conf. Artificial Intelligence, ECAI'94* (Cohn, A.G., Ed.), pp. 307–311. New York: Wiley.

Gero, J., & Kannengiesser, U. (2006). A function–behaviour–structure ontology of processes. *Proc. 2nd Int. Conf. Design Computing and Cognition, DCC'06* (Gero, J.S., Ed.), pp. 407–422. New York: Springer–Verlag.

Goldblatt, R. (1992). *Logics of Time and Computation*, 2nd ed., LNCS, Vol. 7. New York: Springer–Verlag.

Heller, M., & Westfechtel, B. (2003). Dynamic project and workflow management for design processes in chemical engineering. *Proc. 8th Int. Conf. Process Systems Engineering (PSE 2003)*, Kunming, China, June.

Hooker, J.N. (2004). Is design theory possible? *Journal of Information Technology: Theory and Application 6*, 73–82.

Hubka, V., & Eder, W.E. (1988). *Theory of Technical Systems*. Berlin: Springer.

Hubka, V., & Eder, W.E. (1995). *Design Science: Introduction to the Needs, Scope and Organization of Engineering Design Knowledge*. New York: Springer–Verlag.

Jonker, C.M., & Treur, J. (2002). Compositional verification of multi-agent systems: a formal analysis of pro-activeness and reactiveness. *International Journal of Cooperative Information Systems 11*, 51–92.

Jonker, C.M., & Treur, J. (2003). Modelling the dynamics of reasoning processes: reasoning by assumption. *Cognitive Systems Research Journal 4*, 119–136.

Jonker, C.M., Treur, J., & Wijngaards, W.C.A. (2002). Requirements specification and automated evaluation of dynamic properties of a component-based design. *Proc. 7th Int. Conf. AI in Design, AID'02* (Gero, J., Ed.), pp. 547–570. New York: Kluwer Academic.

Jonker, C.M., Treur, J., & Wijngaards, W.C.A. (2003). A temporal modelling environment for internally grounded beliefs, desires and intentions. *Cognitive Systems Research Journal 4(3)*, 191–210.

Kern, C., & Greenstreet, M.R. (1999). Formal verification in hardware design: a survey. *ACM Transactions on Design Automation of Electronic Systems 4(2)*, 123–193.

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

Manzano, M. (1996). *Extensions of First Order Logic*. New York: Cambridge University Press.

Mavris, D.N., Bandte, O., & DeLaurentis, D.A. (1999). Robust design simulation: a probabilistic approach to multidisciplinary design. *AIAA Journal of Aircraft 36(1)*, 298–307.

McMillan, K.L. (1993). *Symbolic model checking: an approach to the state explosion problem*. PhD Thesis. New York: Kluwer Academic.

Nagai, Y., & Taura, T. (2006). Formal description of concept-synthesizing process for creative design. *Proc. 2nd Int. Conf. Design Computing and Cognition, DCC'06* (Gero, J.S., Ed.), pp. 443–460. New York: Springer–Verlag.

Reich, Y. (1995). A critical review of General Design Theory. *Research in Engineering Design 7*, 1–18.

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

Sharpanskykh, A., & Treur, J. (2005). *Verifying Interlevel Relations within Multi-Agent Systems: Formal Theoretical Basis.* Technical Report TR-1701AI, Vrije Universiteit, Amsterdam. Accessed at http://hdl.handle.net/1871/9777

Sharpanskykh, A., & Treur, J. (2006). Verifying interlevel relations within multi-agent systems. *Proc. 17th European Conf. Artificial Intelligence, ECAI'06*, pp. 290–294. New York: IOS Press.

Smithers, T. (1996). On knowledge level theories of design process. *Proc. 4th Int. Conf. Artificial Intelligence in Design, AID'96* (Gero, J.S., & Sudweeks, F., Eds.), pp. 561–579. New York: Kluwer.

Smithers, T. (1998). KLDE—a knowledge level theory of design process. *Proc. 5th Int. Conf. Artificial Intelligence in Design, AID'98* (Gero, J.S., & Sudweeks, F., Eds.), pp. 3–21. New York: Kluwer.

Sosa, R., & Gero, J.S. (2005). A computational study of creativity in design. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing 19(4)*, 229–244.

Thomas, W. (1990). Automata on infinite objects. In *Handbook of Theoretical Computer Science: Formal Models and Semantics* (van Leeuwen, J., Ed.), Vol. B, pp. 133–191. Cambridge, MA: MIT Press.

Tomiyama, T. (1994). From General Design Theory to knowledge intensive engineering. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing 8*, 319–333.

Tomiyama, T., & Yoshikawa, H. (1985). Extended general design theory. In *Design Theory for CAD* (Yoshikawa, H., & Warman, E.A., Eds.), pp. 95–130. Amsterdam: Elsevier Science.

Treur, J. (1991). A logical framework for design processes. *Intelligent CAD Systems III. Proc. 3rd Eurographics Workshop on Intelligent CAD Systems* (ten Hagen, P.J.W., & Veerkamp, P.J., Eds.), pp. 3–20. New York: Springer–Verlag.

van Benthem, J.F.A.K. (1983). *The Logic of Time: A Model-Theoretic Investigation Into the Varieties of Temporal Ontology and Temporal Discourse*. Dordrecht: Reidel.

Warfield, J.N. (1994). *A Science of Generic Design: Managing Complexity Through Systems Design*. Ames, IA: Iowa State University Press.

Yoshikawa, H. (1981). General design theory and a CAD system. *Man–Machine Communication in CAD/CAM, Proc. IFIP Working Group 5.2 Working Conf. 1980* (Sata, T., & Warman, E.A., Eds.), pp. 35–58. Amsterdam: North-Holland.

Tibor Bosse has been an Assistant Professor of artificial intelligence in the Agent Systems Research Group at Vrije Universiteit Amsterdam (VUA) since June 2006. In 2005 he obtained his PhD in artificial intelligence on the topic "Analysis of the Dynamics of Cognitive Processes." Since 2002 his research has been situated at the intersection of artificial intelligence, cognitive science, and ambient intelligence. His main research interest is computational modeling of human-directed (e.g., cognitive, biological, and social) processes for theoretical and practical purposes. Dr. Bosse is a coorganizer of the series of international workshops on human aspects in ambient intelligence.

Catholijn M. Jonker has been a Professor in human–computer interaction and artificial intelligence at the Delft University of Technology since 2006. From 1995 to 2004 she was an Assistant Professor and Associate Professor in the Department of Artificial Intelligence at VUA, and she then held a position as a Professor in artificial intelligence and cognitive science at Radboud University Nijmegen for 2 years. She received her PhD degree in computer science in 1994 from Utrecht University. Dr. Jonker's research focuses on the design and analysis of agent systems and their applications to information agents and electronic commerce. The current general theme of her research interests is the dynamics of the behavior of multiple agents (human and software) in a dynamic environment.

Jan Treur has been a Professor of artificial intelligence at VUA since 1990. He is the Head of the Department of Artificial Intelligence, consisting of about 45 researchers. Prof. Treur is an internationally recognized expert in agent technology, cognitive modeling, and knowledge engineering. He has been a member of the program committees of many of the main conferences and workshops and many journals in these areas. His extensive list of publications covers major scientific publication media in artificial intelligence and cognitive science, including the top level conferences and journals. Some of his recent involvements are organizing and chairing the series of international workshops on human aspects in ambient intelligence. Dr. Treur initiated and designed a strongly multidisciplinary bachelor study program in human ambience at VUA, combining subjects from artificial intelligence, computer science, psychology, and biomedical sciences.

APPENDIX A: DYNAMIC PROPERTIES FOR THE CASE STUDY ON A MULTIAGENT SYSTEM FOR INFORMATION GATHERING

This appendix provides formal definitions of the dynamic properties that are used in the case study (see Section 5). In addition, for some of the interlevel relationships depicted in Figure 2, proof sketches are given. The reader is advised to consult Figure 2 frequently.

A.1. Global properties

DODGP Successfulness. For any trace of the system, there exists a point in time such that in this trace at that point in time agent C will receive a correct conclusion, either from A or from B (or from both).

$$\forall\gamma, \exists t, X, o: [\ \text{state}(\gamma(C), t) \models \text{communicated\_by}(\text{object\_type}(o), \text{true}, X)\ \&\ \text{correct\_value\_for}(\text{object\_type}(o), \text{true})]$$

DODGP1. For each type of information there is a component that initiates information acquisition.

$$\forall v, \gamma, \exists X, t: [\ \text{state}(\gamma(X), t) \models \text{to\_be\_observed}(v)]$$

where $\gamma(X)$ denotes that part of the trace that corresponds to the states of X.

DODGP2. For each type of information, if there is a component that initiates information acquisition, then the external world (E) will generate the relevant information.

$$\forall v, X, \gamma, t: [\ \text{state}(\gamma(X), t) \models \text{to\_be\_observed}(v) \Rightarrow \exists t', s: \text{state}(\gamma(E), t') \models \text{observation\_result\_for}(v, s, X)\ \&\ \text{correct\_value\_for}(v, s)]$$

DODGP3. If E generates correct information of each type, then C will receive a correct conclusion.

$$\forall\gamma, [\forall X, s, v, \exists t: [\ \text{state}(\gamma(E), t) \models \text{observation\_result\_for}(v, s, X)\ \&\ \text{correct\_value\_for}(s, v)\ ] \Rightarrow \exists X, t', o: [\ \text{state}(\gamma(C), t') \models \text{communicated\_by}(\text{object\_type}(o), \text{true}, X)\ \&\ \text{correct\_value\_for}(\text{object\_type}(o), \text{true})]\ ]$$

Note that for the case study, only two types of information are required to draw a conclusion about the object type. The formalization allows for more complex situations in which more information might be needed to draw a conclusion.

A.2. Theorem DODGP1 & DODGP2 & DODGP3 ⇒ DODGP

From DODGP1 we get $\text{state}(\gamma(X), t) \models \text{to\_be\_observed}(v)$, which we can use in DODGP2 (with the correct substitutions) $\text{state}(\gamma(E), t') \models \text{observation\_result\_for}(v, s, X)\ \&\ \text{correct\_value\_for}(v, s)$ and given that we did not need any specific information about v, s, or X, we can deduce the precondition of DODGP3 for the chosen instantiation of γ. Thus, we can deduce by modus ponens that

$$\exists X, t', o: [\ \text{state}(\gamma(C), t') \models \text{communicated\_by}(\text{object\_type}(o), \text{true}, X)\ \&\ \text{correct\_value\_for}(\text{object\_type}(o), \text{true})]$$

Because we made no specific conditions on γ, we can deduce that the statement does in fact hold for all traces.

A.3. Interaction properties

DODI1(X). If X initiated information acquisition, then E will provide the required information for X.

$$\forall v, \gamma, t: [\ \text{state}(\gamma(X), t) \models \text{to\_be\_observed}(v) \Rightarrow \exists t', s: \text{state}(\gamma(E), t') \models \text{observation\_result\_for}(v, s, X)\ \&\ \text{correct\_value\_for}(v, s)]$$

Note that this property corresponds to DODGP2 in the form of a scheme, with variable X.

DODI2(X, Y). If *E* provides information of type IT for *Y*, then *Y* will communicate this information to *X*.

$$\forall\gamma, [\forall X, s, v, \exists t: [\ \text{state}(\gamma(E), t) \models \text{observation\_result\_for}(v, s, Y)\ \&\ \text{correct\_value\_for}(s, v)\ ] \Rightarrow \exists X, t', o: [\ \text{state}(\gamma(Y), t') \models \text{to\_be\_communicated\_to}(v, s, X)\ \&\ \text{correct\_value\_for}(v, s)]\ ]$$

DODI3(X, Y, Z). If *Y* communicated its information of type IT1 to *X* and *E* provided information of the different type IT2 for *X*, then *X* will generate a conclusion on the object and communicate it to *C*.

$$\forall s1, s2, v1, v2, IT1, IT2, t1, t2: [\ \text{state}(\gamma(Y), t1) \models \text{to\_be\_communicated\_to}(v1, s1, X)\ \&\ \text{type}(v1, IT1)\ \&\ \text{state}(\gamma(E), t2) \models \text{observation\_result\_for}(v2, s2, X)\ \&\ \text{type}(v2, IT2) \Rightarrow \exists t', o: [\ \text{state}(\gamma(X), t') \models \text{to\_be\_communicated\_to}(\text{object\_type}(o), \text{true}, C)\ \&\ \text{correct\_value\_for}(\text{object\_type}(o), \text{true})]\ ]$$

Note that this is also a schematic property in which *X*, *Y*, and γ can be instantiated.

A.4. Behavioral properties for a component *X*

DODBP1(X). Here, X is information acquisition proactive.

$$\forall\gamma, t, v: [\ \text{state}(\gamma(X), t) \models \neg\ \text{belief}(X, v, \text{true})\ \&\ \text{state}(\gamma(X), t) \models \neg\ \text{belief}(X, v, \text{false})\ \&\ \text{state}(\gamma(X), t) \models \text{observable\_for}(v, X) \Rightarrow \exists t' > t\ \text{state}(\gamma(X), t') \models \text{to\_be\_observed}(v)\ ]$$

This expresses that whenever *X* has no beliefs about some property *v*, and thinks that component *Y* might know about *v*, then sometime later *X* will initiate a request to *Y* to learn about *v*.

DODBP2(X). Here, *X* is request proactive.

$$\forall X, Y \neq X, \gamma, t, v: [\ \text{state}(\gamma(X), t) \models \neg\ \text{belief}(X, v, \text{true})\ \&\ \text{state}(\gamma(X), t) \models \neg\ \text{belief}(X, v, \text{false})\ \&\ \text{state}(\gamma(X), t) \models \text{belief}(\text{might\_know\_about}(Y, v), \text{true}) \Rightarrow \exists t' > t\ \text{state}(\gamma(X), t') \models \text{to\_be\_communicated\_to}(\text{request}, v, Y)\ ]$$

This expresses that whenever *X* has no beliefs about some property *v*, and thinks that component *Y* might know about *v*, then sometime later *X* will initiate a request to *Y* to learn about *v*.

DODBP3(X). Here, *X* is conclusion proactive.

$$\forall X, \gamma, t, o: [\ \text{entails}(\text{state}(\gamma(X), t), \text{object\_type}(o)) \Rightarrow \exists t' > t\ \text{state}(\gamma(X), t') \models \text{belief}(X, \text{object\_type}(o), \text{true})\ ]\ \&\ [\ \text{entails}(\text{state}(\gamma(X), t), \neg\ \text{object\_type}(o)) \Rightarrow \exists t' > t\ \text{state}(\gamma(X), t') \models \text{belief}(X, \text{object\_type}(o), \text{false})\ ]$$

This expresses that whenever *X* has enough beliefs to infer something about the object type (i.e., the classification of the object at hand), then sometime later *X* will have the corresponding belief about the object type.

DODBP4(X). Here, *X* is information acquisition reactive.

$$\forall X, Y \neq X, \gamma, t, v: [\ \text{state}(\gamma(X), t) \models \neg\ \text{belief}(X, v, \text{true})\ \&\ \text{state}(\gamma(X), t) \models \neg\ \text{belief}(X, v, \text{false})\ \&\ \text{state}(\gamma(X), t) \models \text{belief}(\text{observable\_for}(v, X), \text{true})\ \&\ \text{state}(\gamma(X), t) \models \text{requested}(v, Y) \Rightarrow \exists t' > t\ \text{state}(\gamma(X), t') \models \text{to\_be\_observed}(v)\ ]$$

This expresses that whenever *X* has no beliefs about observable property *v*, and he is requested information about *v*, then some time later *X* will initiate an observation activity to learn about *v*.

DODBP5(X). Here, *X* is information provision reactive.

$$\forall X, Y \neq X, \gamma, t, v, s: [\ \text{state}(\gamma(X), t) \models \text{belief}(X, v, s)\ \&\ \text{state}(\gamma(X), t) \models \text{requested}(v) \Rightarrow \exists t' > t\ \text{state}(\gamma(X), t') \models \text{to\_be\_communicated\_to}(v, s, Y)\ ]$$

This expresses that whenever *X* has believes something about property *v*, and *Y* has requested him information about *v*, then sometime later *X* will communicate what he knows about *v* to *Y*.

DODBP6(X). Here, *X* is information provision proactive.

$$\forall X, Y \neq X, \gamma, t, v, s: [\ \text{state}(\gamma(X), t) \models \text{belief}(X, v, s) \Rightarrow \exists t' > t\ \text{state}(\gamma(X), t') \models \text{to\_be\_communicated\_to}(v, s, Y)\ ]$$

This expresses that whenever *X* believes something about property *v*, then sometime later *X* will communicate what he knows about *v* to everybody else.

A.5. Properties for component *E*

DODEP(X). Initiation of information acquisition by *X* leads to making available the required information for *X*.

$$\forall v, X, \gamma, t: [\ \text{state}(\gamma(E), t) \models \text{to\_be\_observed\_for}(v, X) \Rightarrow \exists t', s: \text{state}(\gamma(E), t') \models \text{observation\_result\_for}(v, s, X)\ \&\ \text{correct\_value\_for}(v, s)]$$

A.6. Transfer properties

DODTP(X, Y). Communication generated by *X* for *Y* is received by *Y*.

$$\forall X, Y \neq X, \gamma, t, v, s: [\ \text{state}(\gamma(X), t) \models \text{to\_be\_communicated\_to}(v, s, Y) \Rightarrow \exists t' > t\ \text{state}(\gamma(Y), t') \models \text{communicated\_by}(v, s, X)\ ]$$

DODTP(X, E). Initiated information acquisition by agent *X* is received by component *E*.

$$\forall v, X, \gamma, t: [\ \text{state}(\gamma(X), t) \models \text{to\_be\_observed}(v) \Rightarrow \exists t' > t\ (\text{state}(\gamma(E), t') \models \text{to\_be\_observed\_for}(v, X)\ ]$$

DODTP(E, X). Information made available by *E* for *X* is received by *X*.

$$\forall v, X, \gamma, t: [\ \text{state}(\gamma(E), t) \models \text{observation\_result\_for}(v, s, X) \Rightarrow \exists t' > t\ (\text{state}(\gamma(X), t') \models \text{observation\_result}(v, s)\ ]$$

A.7. Properties for branches

These properties for branches have only been added for the convenience of the reader and are therefore not formalized.

B1. For each type of information, there is a component that proactively initiates information acquisition.

B2. For information of type IT1 there is a component that proactively initiates information acquisition, and for information of the different type IT2 there is a component that proactively requests the information and another component that reactively initiates information acquisition.

B3. For each type of information, there is a component that proactively requests the information and another component that reactively initiates information acquisition.

B4. If E provides information of type IT for *Y*, then *Y* will proactively provide this information for *X*.

B5. If *E* provides information of type IT for *Y*, then *Y* will reactively provide this information for *X*.

A.8. Proof of DODGP1

We have to prove that for each type of information there is a component that initiates information acquisition. Formally,

$$\forall \text{v}, \gamma, \exists \text{X}, \text{t: [state}(\gamma(\text{X}), \text{t}) \models \text{to\_be\_observed(v)]}$$

Note that this property is proved by three disjuncts (see Fig. 2). Because each of the agents has only the capability to observe the object to be classified from one dimension, both agents need to supply some information. The disjunction in Figure 2, comes from the fact that various configurations ensure that all necessary information is acquired. For this it is easiest to read properties B1, B2, and B3. Note that with less proactiveness in the agents, not enough observations will be made by them. The proof of this part is tedious but easy and thus left to the reader.

It remains to be proven that in each of the subcases property DODGP1 will be satisfied.

Consider case B1, and assume that DODBP1(A) and DODBP1(B) are given. That is, both agents are information acquisition proactive:

$$\forall \gamma, \text{t, v: [state}(\gamma(\text{X}), \text{t}) \models \neg \text{ belief(X, v, true) \& state}(\gamma(\text{X}), \text{t}) \models \neg \text{ belief(X, v, false) \& state}(\gamma(\text{X}), \text{t}) \models \text{observable\_for (v, X)} \Rightarrow \exists \text{t}' > \text{t state}(\gamma(\text{X}), \text{t}') \models \text{to\_be\_observed(v)]}$$

The predicate observable_for(v, X) refers to the different viewing capabilities of the agents. Suppose that *X* can only observe information of some type IT1, then for all possible aspects of the object that are of type IT1, *X* will either already have the relevant information or obtain it by observation. As both agents initially have no information on the object, the effect is that both will obtain all information they possibly can observe. As there are only two information types in the domain, and agents *A* and *B* each observe one type, all possible information is indeed obtained.

To proof B2, one has to consider that one of the agents, say *X*, acquires all necessary information of the type it can observe, say type IT1 [because of property DODBP1(*X*)]. Furthermore, *X* is request proactive, meaning that if it lacks information that another agent, say *Y*, can secure for it, then *X* will request the information from *Y*. Because *Y* has access to information from type IT2, we would now get all relevant information under the following conditions. The request from *X* to *Y* must arrive at *Y* [see property DODTP(*X*, *Y*)], *Y* must be willing to make observations upon request [see property DODBP(4)].

The proof of B3 is similar, although more information exchange is entailed. In this case none of the agents starts to acquire information proactively, but both are proactive in asking the other for information, and both are reactive in making observations (i.e., willing to make them upon request). Of course, also in this case the communication between the two must work correctly.

The proofs of the remaining properties are of the same complexity and as the proofs already given, and left out for reasons of page limitation.

APPENDIX B: LEADS TO SPECIFICATION

B.1. Sorts

property: { dodgp, dodgp1, dodgp2, dodgp3, dodi1(a), dodi1(b), dodi2(x, y), dodi3(x, y, c), dodbp1(x), dodbp1(a), dodbp1(b), dodbp2(x), dodbp2(a), dodbp2(b), dodbp3(x), dodbp4(y), dodbp4(a), dodbp4(b), dodbp5(y), dodbp6(y), dodtp(a, b), dodtp(b, a), dodtp(a, e), dodtp(b, e), dodtp(x, y), dodtp(y, x), dodtp(e, x), dodtp(e, y), dodtp(x, c), dodep(a), dodep(b), dodcheap}
nonlocalproperty: { dodgp, dodgp1, dodgp2, dodgp3, dodi1(a), dodi1(b), dodi2(x, y), dodi3(x, y, c) }
localproperty: { dodbp1(x), dodbp1(a), dodbp1(b), dodbp2(x), dodbp2(a), dodbp2(b), dodbp3(x), dodbp4(y), dodbp4(a), dodbp4(b), dodbp5(y), dodbp6(y), dodtp(a, b), dodtp(b, a), dodtp(a, e), dodtp(b, e), dodtp(x, y), dodtp(y, x), dodtp(e, x), dodtp(e, y), dodtp(x, c), dodep(a), dodep(b), dodcheap}
branch: { b1, b2, b3, b4, b5, b10, b11, b12, b13, b14, b15, b16}
/* Note: branch b10-b16 are "fictive" branch names that are used when there is only one option */
component: {c1a, c1b, c2a, c2b, c3a, c3b, c4a, c4b, c5a, c5b, ew, l1, l2, l3, l4}
dod: {dod(1), dod(2), dod(3), . . .}

B.2. Facts

is_a_subrequirement_of_via(dodgp1, dodgp, b11)
is_a_subrequirement_of_via(dodgp2, dodgp, b11)
is_a_subrequirement_of_via(dodgp3, dodgp, b11)
is_a_subrequirement_of_via(dodbp1(a), dodgp1, b1)
is_a_subrequirement_of_via(dodbp1(b), dodgp1, b1)
is_a_subrequirement_of_via(dodbp1(x), dodgp1, b2)
is_a_subrequirement_of_via(dodbp2(x), dodgp1, b2)
is_a_subrequirement_of_via(dodbp4(y), dodgp1, b2)
is_a_subrequirement_of_via(dodtp(x, y), dodgp1, b2)
is_a_subrequirement_of_via(dodbp2(a), dodgp1, b3)
is_a_subrequirement_of_via(dodbp4(a), dodgp1, b3)
is_a_subrequirement_of_via(dodbp2(b), dodgp1, b3)
is_a_subrequirement_of_via(dodbp4(b), dodgp1, b3)
is_a_subrequirement_of_via(dodtp(a, b), dodgp1, b3)
is_a_subrequirement_of_via(dodtp(b, a), dodgp1, b3)
is_a_subrequirement_of_via(dodi1(a), dodgp2, b12)
is_a_subrequirement_of_via(dodi1(b), dodgp2, b12)

is_a_subrequirement_of_via(dodi2(x, y), dodgp3, b13)
is_a_subrequirement_of_via(dodi3(x, y, c), dodgp3, b13)
is_a_subrequirement_of_via(dodep(a), dodi1(a), b14)
is_a_subrequirement_of_via(dodtp(a, e), dodi1(a), b14)
is_a_subrequirement_of_via(dodep(b), dodi1(b), b15)
is_a_subrequirement_of_via(dodtp(b, e), dodi1(b), b15)
is_a_subrequirement_of_via(dodbp6(y), dodi2(x, y), b4)
is_a_subrequirement_of_via(dodtp(e, y), dodi2(x, y), b4)
is_a_subrequirement_of_via(dodbp2(x), dodi2(x, y), b5)
is_a_subrequirement_of_via(dodbp5(y), dodi2(x, y), b5)
is_a_subrequirement_of_via(dodtp(x, y), dodi2(x, y), b5)
is_a_subrequirement_of_via(dodtp(e, y), dodi2(x, y), b5)
is_a_subrequirement_of_via(dodbp3(x), dodi3(x, y, c), b16)
is_a_subrequirement_of_via(dodtp(y, x), dodi3(x, y, c), b16)
is_a_subrequirement_of_via(dodtp(e, x), dodi3(x, y, c), b16)
is_a_subrequirement_of_via(dodtp(x, c), dodi3(x, y, c), b16)

costs(c1a, 500)
costs(c1b, 501)
costs(c2a, 350)
costs(c2b, 351)
costs(c3a, 120)
costs(c3b, 121)
costs(c4a, 90)
costs(c4b, 91)
costs(c5a, 50)
costs(c5b, 51)
costs(ew, 0)

predicted_costs(b1, 60)
predicted_costs(b2, 300)
predicted_costs(b3, 900)
predicted_costs(b4, 200)
predicted_costs(b5, 500)
predicted_costs(b11, 700)
predicted_costs(b12, 350)
predicted_costs(b13, 350)
predicted_costs(b14, 150)
predicted_costs(b15, 150)
predicted_costs(b16, 200)

quality(c1a, 100)
quality(c1b, 100)
quality(c2a, 250)
quality(c2b, 250)
quality(c3a, 50)
quality(c3b, 50)
quality(c4a, 400)
quality(c4b, 400)
quality(c5a, 150)
quality(c5b, 150)
quality(ew, 1000)

B.3. Local properties

Note that the timing parameters *e*, *f*, *g*, and *h* have been left out in these local properties. However, they can be filled in at will. For example, by setting them to {0, 0, 1, 1} for LP0, this would imply that immediately after the start of the system (i.e., with a delay of zero time units), the initial requirements are derived for one time unit.

B.4. Requirements

B.4.1. LP0 initialization

The first local property LP0 expresses that the initial requirements for the system are dodgp and dodcheap. Formalization:

start ↠ is_a_current_requirement(dodgp)
∧ is_a_current_requirement(dodcheap)

B.4.2. LP1a requirement persistence

Local property LP1a expresses that, once it is derived that a requirement is needed for the system, this requirement will persist, as long as the stakeholder does not indicate that the requirement (or the branch it belongs to) is undesirable. Formalization:

is_a_current_requirement(p) ∧ not(undesirable_requirement(p)) ∧ is_a_subrequirement_of_via(p, q, b) ∧ best_branch_for(b, q) ∧ not(undesirable_branch(b)) ↠ is_a_current_requirement(p)

B.4.3. LP1b refined requirement persistence

Local property LP1b expresses that, once a local requirement has been refined via a certain branch, this will remain the case, and the branch will be marked as "current" branch (needed within LP9). Formalization:

requirement_refined_via(n, b) ∧ not(undesirable_branch (b)) ↠ requirement_refined(n) ∧ requirement_refined_via(n, b) ∧ is_a_current_branch(b)

B.4.4. LP2 undesirable branch determination

These local properties are used to determine which branches are undesirable. There are two cases: a requirement that belongs to it is undesirable and its total costs are higher than predicted while the requirement dodcheap is present. Formalization:

is_a_subrequirement_of_via(p, n, b) ∧ undesirable_requirement(p) ↠ undesirable_branch(b) is_a_current_requirement(dodcheap) ∧ total_branch_costs(b, x) ∧ predicted_costs(b, y) ∧ x > y ↠ undesirable_branch(b)

B.4.5. LP3 branch selection

These five local properties are used to determine which is the best branch (or subtree) that can be selected to satisfy a given nonlocal requirement. In case the requirement dodcheap is present, the selection criterion is the predicted costs: the branch of which these are the lowest is selected. However, other criteria (such as the quality) can be implemented as well. Furthermore, we have to check whether the stakeholder has not indicated that the branch is undesirable. Formalization:

is_a_subrequirement_of_via(p, n, b) ↠ branch_for(b, n)
is_a_current_requirement(dodcheap) ∧ branch_for(a, n) ∧ branch_for(b, n) ∧ not(undesirable_branch(a)) ∧ predicted_costs(a, x) ∧ predicted_costs(b, y) ∧ x ≤ y ↠ better_branch_than_for(a, b, n)

branch_for(a, n) ∧ branch_for(b, n) ∧ not(undesirable_
branch(a)) ∧ undesirable_branch(b) ↠
better_branch_than_for(a, b, n)
branch_for(a, n) ∧ not(branch_for(b, n))
↠ better_branch_than_for(a, b, n)
[∀b:branch better_branch_than_for(a, b, n)]
↠ best_branch_for(a, n)

B.4.6. LP4 requirement refinement

Local property LP4 expresses that, if a requirement exists that can be refined to a subrequirement, then this should be done by refining via the best branch (e.g., the one with the lowest costs, see LP3). Formalization:

is_a_current_requirement(p) ∧ is_a_subrequirement_of_
via(q, p, b) ∧ not(requirement_refined(p))
∧ best_branch_for(b, p) ∧ not(undesirable_branch(b))
↠ is_a_current_requirement(q) ∧ requirement_
refined(p) ∧ requirement_refined_via(p, b)

B.5. DODs

B.5.1. LP5 component selection

These four local properties are used to determine which is the best component that can be selected in order to satisfy a given requirement. Again, when the requirement dodcheap is present, the selection criterion is the price: the component that satisfies the requirement with the lowest costs is selected. Furthermore, we have to check whether the stakeholder has not indicated that the component is undesirable. Formalization:

is_a_current_requirement(dodcheap) ∧ holds_for(l, c)
∧ holds_for(l, d) ∧ not(undesirable_component(c))
∧ costs(c, x) ∧ costs(d, y) ∧ x ≤ y ↠ better_component_
than_for(c, d, l)
holds_for(l, c) ∧ holds_for(l, d) ∧ not(undesirable_component(c))
∧ undesirable_component(d) ↠ better_component_
than_for(c, d, l)
holds_for(l, c) ∧ not(holds_for(l, d)) ↠ better_component_
than_for(c, d, l)
[∀d:comp better_component_than_for(c, d, l)]
↠ best_component_for(c, l)

B.5.2. LP6 DOD generation

This property expresses that, if DODM (i.e., a specific module for DOD generation and manipulation) is active, then each local requirement should be satisfied by adding the best component for that requirement to the current DOD. Moreover, the costs of that component should be stored as "intermediate branch costs" (needed by LP10). Formalization:

"DODM_active" ∧ is_a_current_requirement(l)
∧ best_component_for(c, l) ∧ not(undesirable_
component(c)) ∧ costs(c, y) ∧ is_a_subrequirement_
of_via(l, n, b) ∧ "DOD_counter"(x)
↠ current_DOD(dod(x)) ∧ part_of_DOD(c, dod(x))
∧ intermediate_branch_costs(b, y)

B.5.3. LP7a DOD persistence

Local property LP7a expresses that, once a DOD is the current DOD, this will remain the case until the DOD_counter has been increased. Formalization:

current_DOD(dod(x)) ∧ "DOD_counter"(x)
↠ current_DOD(dod(x))

B.5.4. LP7b DOD component persistence

Local property LP7b expresses that, once a certain component has been added to a DOD, it will remain part of that DOD forever. Formalization:

part_of_DOD(c, d) ↠ part_of_DOD(c, d)

B.5.5. LP8 requirement satisfaction determination

This property determines when a certain (local) requirement is satisfied by a DOD. This is the case when the current DOD contains a component for which this requirement holds. Formalization:

current_DOD(d) ∧ part_of_DOD(c, d) ∧ holds_for(l, c)
∧ is_a_current_requirement(l) ↠ local_
requirement_satisfied(l)

B.5.6. LP9 requirement "dodcheap" satisfaction determination

These properties determine when the requirement that "the costs of the branches should not be higher than the predicted costs" is satisfied by a DOD. This is only the case if, for each branches, it is not a current branch OR it is not a "leaf" branch OR its costs do not exceed the predicted costs. Formalization:

"DODM_active" ∧ not(is_a_current_branch(b))
↠ branch_covered(b)
"DODM_active" ∧ is_a_current_branch(b)
∧ is_a_subrequirement_of_via(p, q, b)
↠ branch_covered(b)
"DODM_active" ∧ is_a_current_branch(b)
∧ total_branch_costs(b, x) ∧ predicted_costs(b, y) ∧ x ≤ y
↠ branch_covered(b)
[∀b:branch branch_covered(b)]
↠ local_requirement_satisfied(dodcheap)

B.5.7. LP10 total branch costs calculation

These properties are used to calculate the total costs for a certain branch. To do this, all *x* for which intermediate_branch_costs(b, x) holds should be added. This is done by giving the lowest *x* the index 0, giving the next *x* the index 1, and so on. Next, all *x* are added stepwise. Note that this approach assumes that each component has a different price. Formalization:

intermediate_branch_costs(b, x) ∧ intermediate_branch_
costs(b, y) ∧ x > y ↠ not_index(b, 0, x)
intermediate_branch_costs(b, x) ∧ intermediate_branch_
costs(b, y) ∧ x > y ∧ not_index(b, I, x) ∧ not_index(b, I, y)
↠ not_index(b, I + 1, x)
intermediate_branch_costs(b, x) ∧ intermediate_branch_
costs(b, y) ∧ x < y ∧ not_index(b, I, y) ∧ not(not_index
(b, I, x)) ∧ not(hasindex(b, x)) ∧ sum(b, I, s) ↠ index(b, I + 1, x)
∧ hasindex(b, x) ∧ hassum(b, I + 1) ∧ sum(b, I + 1, s + x)
intermediate_branch_costs(b, x) ↠ intermediate_branch_
costs(b, 10000) ∧ intermediate_branch_costs(b, x)

hasindex(b, x) ↠ hasindex(b, x)
hassum(b, x) ↠ hassum(b, x)
index(b, x, y) ↠ index(b, x, y)
sum(b, x, y) ↠ sum(b, x, y)
sum(b, x, y) ∧ not(hassum(b, x + 1))
  ↠ total_branch_costs(b, y)

B.5.8. LP11 total DOD costs calculation

These properties are used to calculate the total costs for the final DOD. To do this, the same algorithm is used as in LP10. Formalization:

DOD_generation_terminated ∧ is_a_current_branch(b)
  ∧ total_branch_costs(b, x) ↠ intermediate_DOD_costs(x)
intermediate_DOD_costs(x) ∧ intermediate_DOD_costs(y)
  ∧ x > y ↠ not_index(0, x)
intermediate_DOD_costs(x) ∧ intermediate_DOD_costs(y)
  ∧ x > y ∧ not_index(I, x) ∧ not_index(I, y)
  ↠ not_index(I + 1, x)
intermediate_DOD_costs(x) ∧ intermediate_DOD_costs(y)
  ∧ x < y ∧ not_index(I, y) ∧ not(not_index(I, x))
  ∧ not(hasindex(x)) ∧ sum(I, s) ↠ index(I + 1, x)
  ∧ hasindex(x) ∧ hassum(I + 1) ∧ sum(I + 1, s + x)
intermediate_DOD_costs(x) ↠ intermediate_DOD_
  costs(10000) ∧ intermediate_DOD_costs(x)
hasindex(x) ↠ hasindex(x)
hassum(x) ↠ hassum(x)
index(x, y) ↠ index(x, y)
sum(x, y) ↠ sum(x, y)
sum(x, y) ∧ not(hassum(x + 1)) ↠ total_DOD_costs(y)

B.5.9. LP12 RQSM activity determination

Local property LP12 expresses that, as long as there are nonlocal requirements that have not been refined yet, RQSM (i.e., a specific module for requirements refinement) is still active. Formalization:

is_a_current_requirement(n) ∧ not(requirement_refined(n))
  ↠ "RQSM_active"

B.5.10. LP13 DODM activity determination

These three properties express that DODM will be active when RQSM is not active but has been active before. Formalization:

"RQSM_active" ↠ "RQSM_earlier_active"
"RQSM_earlier_active" ↠ "RQSM_earlier_active"
"RQSM_earlier_active" ∧ not("RQSM_active")
  ↠ "DODM_active"

B.5.11. LP14 DOD counter

These properties are used to keep track of the index of the current DOD. Initially, the counter is set to 0. It is increased each time that RQSM changes from active to inactive. Formalization:

start ↠ "DOD_counter"(0)
"RQSM_active" ↠ "DOD_counter_ready"
"DOD_counter"(x) ∧ not("RQSM_active")
  ∧ "DOD_counter_ready" ↠ "DOD_counter"(x + 1)
"DOD_counter"(x) ∧ not("DOD_counter_ready")
  ↠ "DOD_counter"(x)
"DOD_counter"(x) ∧ not("RQSM_active")
  ↠ "DOD_counter"(x)

B.5.12. LP15 determination of treated requirements

These three properties express the three situations when a requirement has been "covered" (and thus should not be treated anymore). These situations are when the requirement is not required for the given design object, when the requirement has been refined, and when the requirement is satisfied. Formalization:

"DODM_active" ∧ not(is_a_current_requirement(p))
  ↠ requirement_covered(p)
"DODM_active" ∧ requirement_refined(n)
  ↠ requirement_covered(n)
"DODM_active" ∧ local_requirement_satisfied(l)
  ↠ requirement_covered(l)

B.5.13. LP16 termination determination

Local property LP16 expresses that, as soon as all requirements have been covered, the process will terminate. Formalization:

[∀p:prop requirement_covered(p)]
  ↠ DOD_generation_terminated

APPENDIX C: ADDITIONAL SIMULATION TRACES

C.1. Trace 1 and Trace 2

See Section 8.

C.2. Trace 3

The beginning of this trace is similar to that of Trace 1 and 2 (Fig. C.1). However, at the exact moment that all present requirements have been refined (time point 8) and the system just started to create a first DOD (DOD(1)), the atom undesired_requirement(DODBP1(b)) becomes true (representing the fact that the stakeholder has indicated that this requirement is not desirable). Notice that this is a more radical decision than rejecting only a component. The result is that the process has to switch back to the phase of requirement refinement. Here, "requirement revision" takes place; that is, the undesirable requirement is rejected and its parent is now refined via another requirement. When this has happened, a second attempt is made to generate a satisfying DOD [DOD(2)], on the basis on the new set of requirements. However, note that the information concerning the first DOD [DOD(1)] is not completely deleted. Instead, it is kept in memory, just in case it will be needed in a later stage of the process.

C.3. Trace 4

This is a trace where "branch revision" occurs several times (Fig. C.2). Branch revision happens as follows: the system first chooses a branch and refines it to local requirements. Next, a DOD is created, consisting of an assignment of components to the local requirements. Based on the costs of these components, the system calculates the total costs

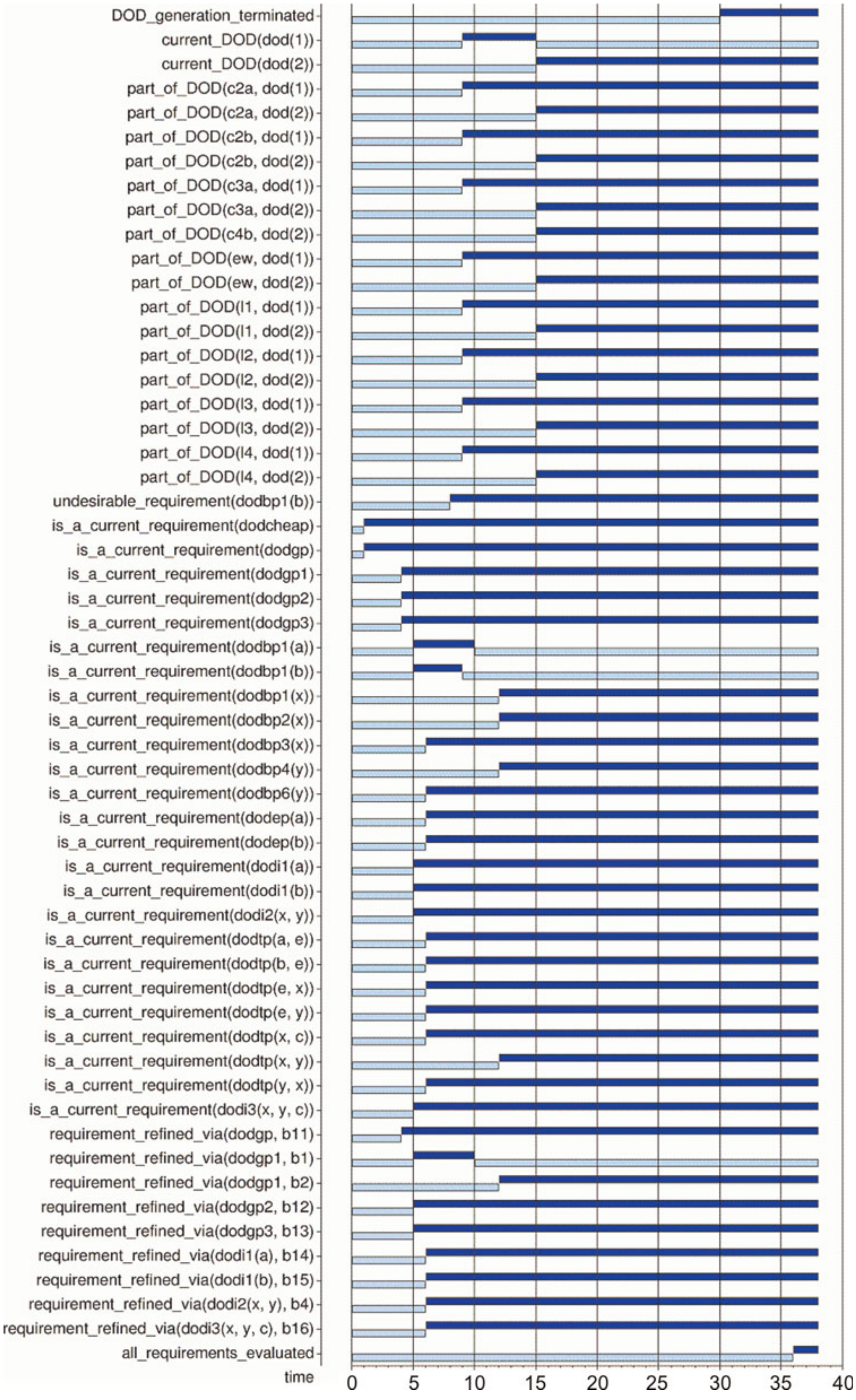

Fig. C.1. Simulation trace 3. [A color version of this figure can be viewed online at journals.cambridge.org/aie]

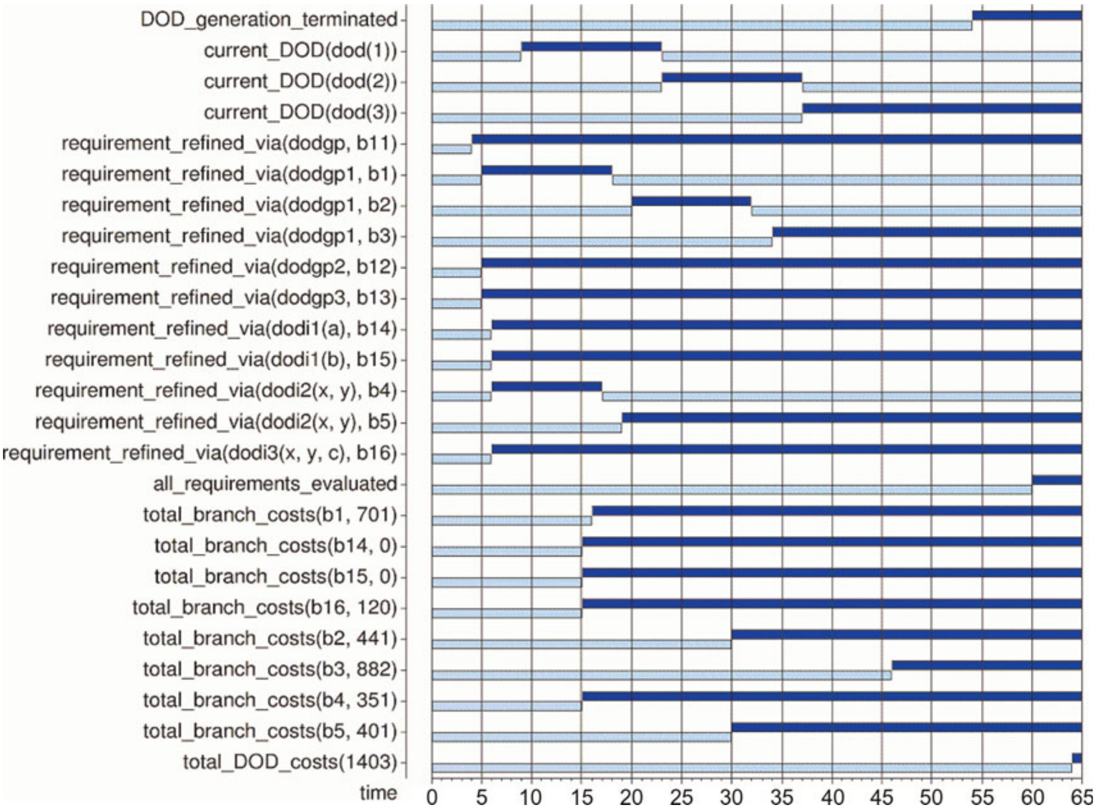

Fig. C.2. Simulation trace 4. [A color version of this figure can be viewed online at journals.cambridge.org/aie]

for that branch. In case this number is higher than the predicted costs for that branch, the branch is marked as undesirable. As a result, the system switches back to the phase of requirements refinement to "backtrack" in the tree and choose another branch. In total, three branches are revised in trace 4 (B1, B4, and B2). Notice that, instead of revising a branch because the costs are too high, there could be several other reasons as well. In the end of the process, the total costs of the resulting DOD are calculated.