Formal Analysis of Design Process Dynamics*

Tibor Bosse[1], Catholijn M. Jonker[2], Jan Treur[1]

[1]Vrije Universiteit Amsterdam, Department of Artificial Intelligence
De Boelelaan 1081a, 1081HV Amsterdam, The Netherlands
URL: http://www.few.vu.nl/~{tbosse, jonker, treur}
Email: {tbosse, jonker, treur}@cs.vu.nl

[2]Radboud University Nijmegen, Nijmegen Institute for Cognition and Information
Montessorilaan 3, 6525 HR Nijmegen, The Netherlands
Email: C.Jonker@nici.ru.nl

Abstract. To enable the development of automated support for the dynamics of design processes, a challenge is to model and analyse such dynamics in a formal manner. This paper contributes a declarative, logical approach for specification of dynamic properties of design processes, supported by a formal temporal language which has a high expressivity. This language can be used to specify dynamic properties at the level of a design process as a whole, or of parts thereof. At the most detailed level, in an executable sublanguage also simulation models are specified in a declarative, logical manner, which allows to use these specifications in logical analysis as well. The approach is illustrated by an example component-based agent-system design process.

1. Introduction

Providing automated support to manage the dynamics of a design process is in most cases far from trivial. For example, in (Corkill 2000) some of the requirements put forward are that (1) a complete design process representation is needed, (2) with sufficient detail to allow for direct execution. Also by (Brown and Chandrasekaran 1989; Heller and Westfechtel 2003; Baldwin and Chung 1995; Gero and Kannengiesser, 2006) it is put forward that supporting the management of the dynamics of a design process is an important challenge to be addressed. This indeed is the aim of the current paper.

The type of design considered is the design of component-based (e.g., software) systems for dynamic applications. In such application areas often components can be (re)used for which the (dynamic) properties are known. By composing a number of such components in a component-based design, the required overall dynamics is obtained. If the dynamics required is not that simple, it is not straightforward how such dynamics relates to available reusable components and their dynamic properties.

As holds for many design processes, designing component-based systems can be a rather complex and dynamic process, for which a number of tasks play a role, for example:

---

* An earlier, shorter report of part of this work was presented at ECAI 2004.

1. maintaining of specifications of properties of (reusable) components
2. maintaining of 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 this 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
6. checking whether (a design object description of) a component-based system, with known properties of its components, satisfies the requirements
7. revision of a design object description that does not fully satisfy the requirements
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 *design object descriptions* (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, Treur, and Wijngaards 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 behaviour that have to be fulfilled
- a partial description of (prescribed) system architecture that has to be incorporated
- a partial description of (prescribed) dynamic properties of elements of the system that have to be incorporated; e.g., for components, for transfers, for parts, for interactions between parts.

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

In this paper, Section 2 a formalisation of design process dynamics will be discussed 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. In Section 5, a relevant requirement will be given for the example system to be designed. It will be shown how this global requirement for the overall system can be 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 will describe a simulation model of the example design process, whereas Section 8 discusses some example simulation traces. In Section 9 the example design process is analysed 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 shows some of the logical relationships between these dynamic properties. Finally, Section 11 is a discussion.

2. Describing Design Process Dynamics

To analyse dynamics of a design process, a formalisation is needed of such dynamics. Such a formalisation is introduced in this section, based on (Treur, 1989; Jonker and Treur, 2002). This formalisation uses the notions of design state and design trace, where design states and design traces are composed of a part for requirements and a part for design object descriptions, an approach adopted from (Treur, 1989); see also (Gavrila and Treur, 1994; Brazier, Langen, Ruttkay, and Treur, 1994; Brazier, Langen and Treur, 1996).

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, $\mathcal{S} = < \mathcal{S}_1, \mathcal{S}_2 >$ with:

$\mathcal{S}_1$	*requirements state*	(including the current requirements set)
$\mathcal{S}_2$	*design object description state*	(including the current design object description)

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

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), i.e., 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 in order to *analyse* 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 and Treur 2002). 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; Fisher, 2005). These languages usually have expressivity limited to prositional logic. The language TTL is briefly defined as follows.

The Language 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). 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 state(γ, t). These states can be related to state properties via the formally defined satisfaction relation denoted by the infix predicate |=; here state(γ, 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. (Reiter, 2001; Kowalski and Sergot, 1986). Notice that here state properties are represented (reified) by terms to denote them as objects in the TTL language. 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 ¬, &, ∨, ⇒, ∀, ∃.

The Language LEADSTO for Executable Dynamic Properties
To be able to perform some automated experiments with design processes, a simpler temporal logical language to specify simulation models has been used. This language LEADSTO enables to model direct temporal dependencies between two state properties in successive states, as occur in specifications of a simulation model (for example, if in the current state, state property p holds, then in the next state, state property q holds). This language is exectuable 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 ground atoms or negations of ground atoms'. In the leads to 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.

For a formal definition of the leads to language in terms (as a sublanguage) of the language TTL, see (Jonker, Treur, and Wijngaards 2003).

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.

Local Properties
Local properties only concern the smallest steps (taken into account in the conceptualisation of the process) in the process under analysis. An example Local Property of a design process might be (simplified, and in semi-formal notation):

At every point in time,
if a requirement **r** is imposed on the object to be designed,
and this requirement can be refined into sub-requirement **q**
then at the next point in time, sub-requirement **q** will be imposed on the object to be designed

Global Properties
In contrast, a global property is a non-local 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 design object description **D** such that, for each requirement **r** in **R**,
the design object description **D** satisfies requirement **r**

More complex Local and Global dynamic properties for design processes and their formalisations 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 behaviour. 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 behaviour, 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 behaviour 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 behaviour are given, this scheme shows that to fulfil these overall dynamic properties, dynamic properties for certain parts and for interaction between these parts are needed that together imply the overall behaviour requirements. The process to identify new, refined requirements for behaviour 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 multi-agent system for cooperative information gathering (Jonker and 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 Figure 1. The resulting behaviour 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 sources of information, but this differs for the two agents. By co-operation they can benefit from the exchange of information that is only accessible to the other agent. If both types of information are combined, conclusions can be drawn that would not have been achievable for each of the agents separately. Why could such a co-operation fail? First of all, one of the agents, say A, may not be pro-active in its individual search for information. This might be compensated if the agent B is pro-active in 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, for example, 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 pro-activeness* and *conclusion pro-activeness* 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 Figure 1. In this figure the ovals denote the three agents. The arrows depict channels for flow of information (socalled information links). Each of the agents is able to acquire partial information from an external source by initiated observations. Initiated observations are modelled 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 co-operative information agents is materialised in the following more concrete form. The world situation consists of an object that has to be classified. One agent can observe only the bottom view of the object, the other agent 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, e.g., if the agent knows that the views are a circle and a square, it is concluded that the object is a cone.

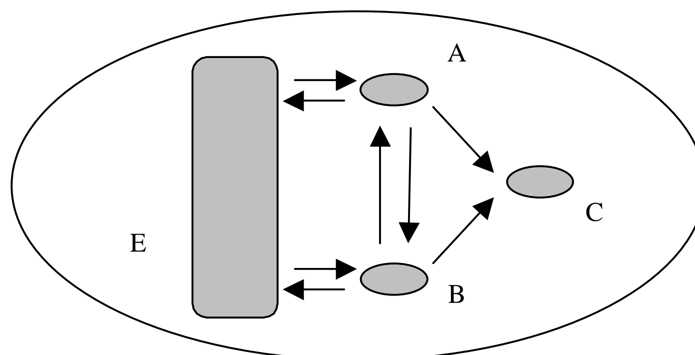

Figure 1. The Example Agent System to be Designed

In most multi-agent 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, for instance, in their attitudes towards observation and communication: an agent may or may not be *pro-active*, in the sense that it takes the initiative with respect to one or more of:

- performing observations
- communicate its own observation results to the other agent
- ask the other agent for its observation results
- 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
- 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 pro-active 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. Or, an agent that is only reactive in reasoning and in information acquisition may come to a conclusion due to pro-activeness of the other agent. So, successfulness can be achieved in many ways and depends on subtle interactions between pro-activeness 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. First, Section 5.1 presents the design problem statement, consisting of the requirements on the overall agent system behaviour, 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 behaviour requirement, interaction dynamics (transfers), and prescribed behaviours 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 DODGP (Design Object Description Global Property):

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 behaviour 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 behaviour 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 behaviour be refined to requirements on agent behaviours? Such a requirements refinement process can be managed more effectively if the overall requirements are not directly related to agent behaviour requirements, but one or more intermediate levels are created. The idea is that for the agent system to be successful it is needed 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 Figure 2) provides the dynamic properties DODGP1, DODGP2, 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 Figure 2) by the implication:

$$\textbf{DODGP1 \& DODGP2 \& DODGP3} \Rightarrow \textbf{DODGP}$$

A next step in the requirements refinement process is to relate each of the dynamic properties DODGP1, DODGP2 and DODGP3 to agent behaviour properties. The complete refinement of

these properties is elaborated in Appendix A. Below, we only present the tree with logical relationships between dynamic properties, without showing the exact definitions of the properties.

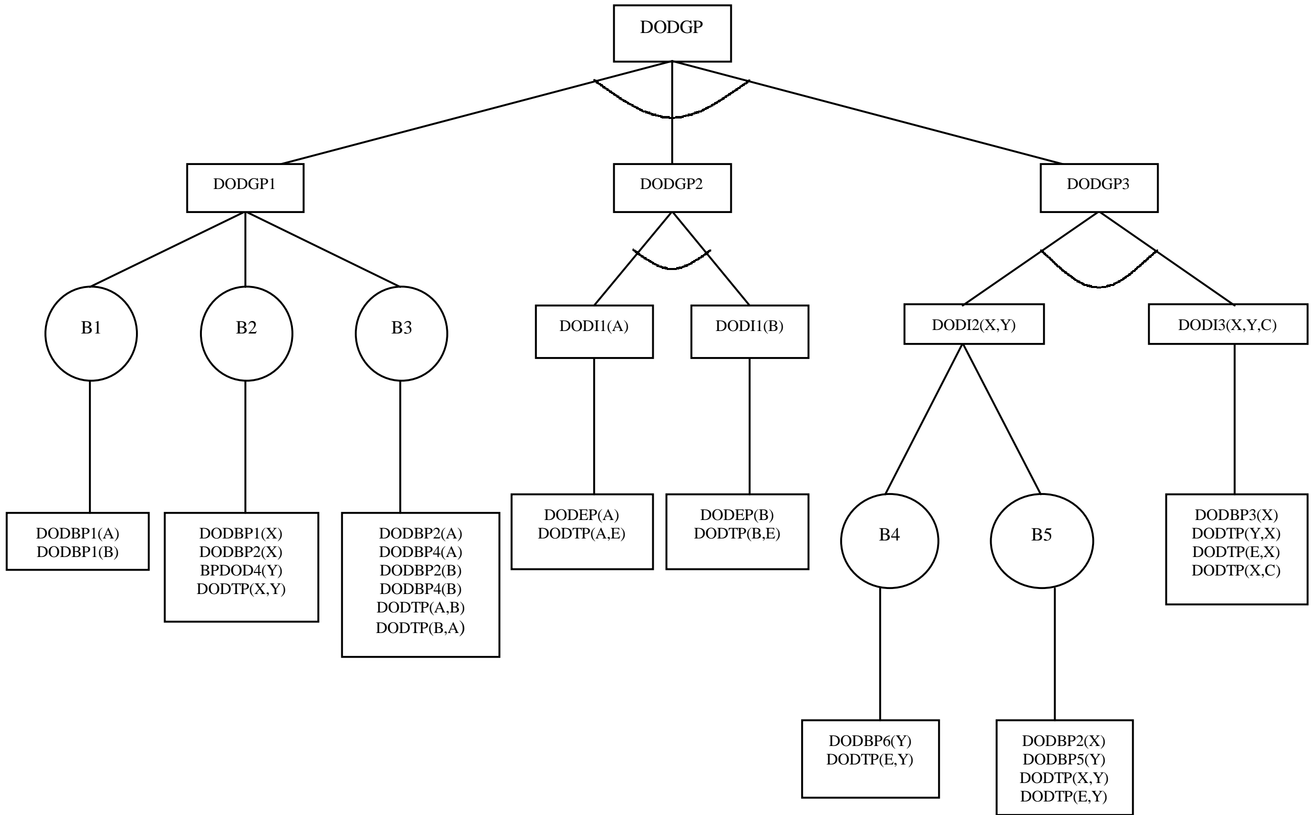

Figure 2. Overview of all possible requirement refinements

In Figure 2, in the form of and AND/OR tree (corresponding to upward logical implications) an overview is shown of all possible refinements as discussed. Here X and Y are variables over the set {agent A, agent B}, where X ≠ Y. Note that the different alternative branches are indicated by the names B1 to B5. Conjunctions of branches are indicated by arcs connecting them.

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.

TABLE 1. Example Design Trace

Step	Informal Desciption	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.	Requirement Set (1): {DODGP}
3	Using refinement knowledge, the requirements are refined into requirements for components and interactions (transfers) between components. 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 design object description state. The library indicates for each component which of the properties it has (e.g., component c2s satisfies property DODBP1(A)).	DOD(1): {c2a, c2b, c3a, ew}
5	The connections are made according to the transfer requirements and included in the design object description state. This can also make use of the library, or 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 design object description, based on the logical relationships.	"All requirements evaluated"

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 Figure 2 and the characteristics of components as stored in the library) have been modelled by means of *sorts* and *facts*. The domain-independent information (e.g., rules that refine a requirement to its sub-requirements) has been modelled by means of *local properties*. However, notice that one of these local properties is domain-specific, namely the initialisation 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 modelled 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
- 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 sub-requirement of property dodgp1 via branch b1 (see the lower-left edge in Figure 3). Notice that, in cases where there is only one branch to choose among, such a (fictive) branch has been included in the formal notation. This differs from the visual notation, where the branch has been left out. An example of such a fact representing the logical relationships between requirements is:

is_a_subrequirement_of_via(dodgp2, dodgp, b11)

During simulation, these logical relationships are used as heuristics in order 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. 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 design object description at the end of the design process. As a guideline for such an evaluation, a similar tree as in Figure 2 is used, but this time the information is complete and bottom-up. 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.

holds_for(dodbp1(a), c1a)	costs(c1a, 500)	quality(c1a, 100)	predicted_costs(b1, 60)
holds_for(dodbp2(a), c1a)	costs(c1b, 501)	quality(c1b, 100)	predicted_costs(b2, 300)
holds_for(dodbp3(a), c1a)	costs(ew, 0)	quality(ew, 1000)	predicted_costs(b3, 900)
holds_for(dodbp4(a), c1a)			

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 Design Object Description states. In this section, only a subset of the Local Properties used for the simulation is shown. The complete specification of the simulation is shown in Appendix B.

Properties concerning requirements

Within the process requirements are determined and refined. This process takes into account whether or not the stakeholder asserts that certain requirements are undesirable.

LP0 Initialisation

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. Formalisation:

start ↠ is_a_current_requirement(dodgp) ∧ is_a_current_requirement(dodcheap)

LP2 Undesirable Branch Determination

These two local properties are used to determine which branches are undesirable. There are two cases: (1) a requirement that belongs to it is undesirable and (2) its total costs are higher than predicted, whilst the requirement dodcheap is present. Formalisation:

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)

LP4 Requirement Refinement

Local property LP4 expresses that, if currently a requirement p exists that can be 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 costs). Formalisation:

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)

Properties concerning the Design Object Description

The process concerning design object descriptions determines design object descriptions 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 requirement to the current design object description dod(x). Formalisation:

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 design object description. This is the case when the current design object description contains a component c for which this requirement holds. Formalisation:

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

8. Some Example Simulation Traces

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

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

The first trace (depicted in Figure 3) shows a design process in which no revision is needed. Time is on the horizontal axis, the derived state properties are on the vertical axis. In this simulation, for all local properties the values (0,0,1,1) have been chosen for the timing parameters e, f, g, and h. To facilitate understanding, only the most relevant of the derived atoms are shown.

When the process starts, first the initial requirements dodgp and dodcheap are identified. After this, these requirements are refined into sub-requirements dodgp1, dodgp2 and dodgp3 (based on the logical relationships of the tree in Figure 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 design object description (called dod(1)) is created which consists of a number of components (and connections between them) that satisfy all local requirements.

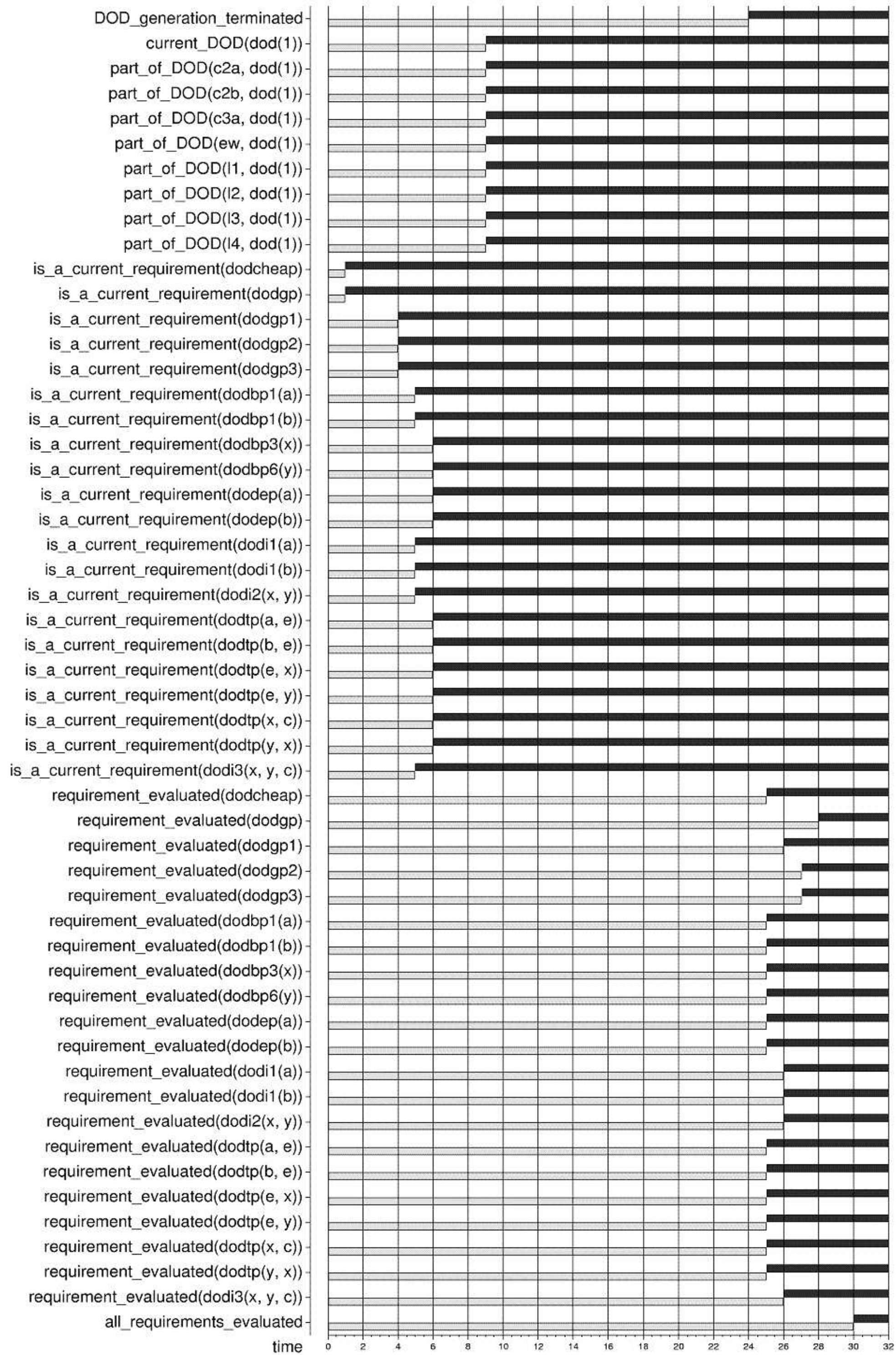

Figure 3. Simulation trace 1

During this process, if possible only the components with the lowest costs are selected. As soon as a satisfactory design object description has been found (at least according to the requirements that were derived), design object description generation finishes, and after this, all (local and nonlocal) requirements that are part of the design object description 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.

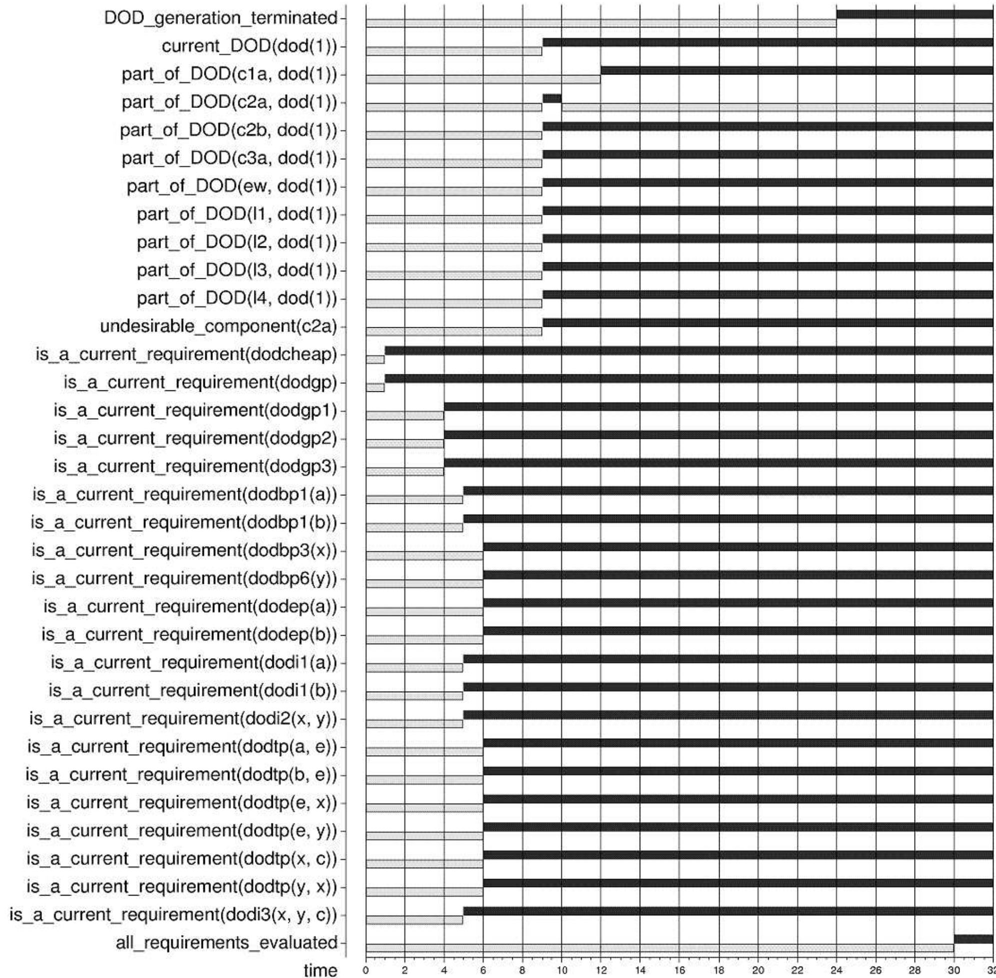

Figure 4. Simulation trace *2*

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 design object description 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 design object description and replaces it by another (probably more expensive) component, c1a. Finally, the design process succeeds in finding a satisfactory design

object description. This resulting design object description is then evaluated and its total costs are calculated. Two other examples of simulation traces are described in Appendix C.

9 Global Dynamic Properties of a Design Process

For design processes like the one described above, a number of global dynamic properties can be identified. For example:

- During (or after termination of) the design process, the design process objectives are fulfilled
- After termination of the design process the final design object description 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 design object description generated during the process solely consists of standard components

In this section a number of such dynamic properties, expressed as TTL statements, are presented. 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.

9.1 Global Dynamic Properties

The following global properties have been identified and formally specified.

GP1 Local Requirement Satisfaction
Eventually there is a design object description that contains a satisfactory component for each local requirement that exists at that moment.
Formalisation:

∃t ∃d:DOD state(γ, t) |= current_DOD(d) & ∀r:localreq [state(γ, t) |= is_a_current_requirement(r) ⇒
∃ c:component state(γ, t) |= part_of_DOD(c, d) & state(γ, t) |= holds_for(r, c)]

GP2 Termination of the Design Process
Eventually the process will terminate.
Formalisation:

∃t state(γ, t) |= DOD_generation_terminated

GP3 Cheapest Components Per Local Requirement
For each local requirement, if there is a component that satisfies it, then the cheapest component that satisfies it will be added to the DOD. (Note that this is a refinement of GP4).
Formalisation:

∀t ∀r:localreq ∀c:component
state(γ, t) |= is_a_current_requirement(r) & state(γ, t) |= holds_for(r, c) ⇒
∃t'≥t ∃d:DOD ∃c':component ∃p:integer
state(γ, t') |= part_of_DOD(c', d) & state(γ, t') |= holds_for(r, c') & state(γ, t') |= costs(c', p) &
¬[∃c'':component ∃p' <p state(γ, t') |= holds_for(r, c'') & state(γ, t') |= costs(c'', p')]

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.

Formalisation:

$\forall t\ \forall r\ \forall$:localreq $\forall c$:component
state(γ,t) |= is_a_current_requirement(r) & state(γ, t) |= holds_for(r, c) ⇒
∃t'≥t ∃d:DOD ∃c':component state(γ, t') |= part_of_DOD(c', d) & state(γ, t') |= holds_for(r, c')

GP5(c) Total Costs

Eventually the system generates a DOD of which the costs are exactly c.

Formalisation:

∃t state(γ,t) |= total_DOD_costs(c)

GP6 Requirement Persistence

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

Formalisation:

∀t ∀r:req state(γ, t) |= is_a_current_requirement(r) ⇒ ∀t'≥t state(γ, t') |= is_a_current_requirement(r)

GP7 New DOD Grounding

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

Formalisation:

∀t ∀t'>t ∀x:integer [state(γ, t) |= current_DOD(x) & state(γ, t') |= current_DOD(x+1)]
⇒ ∃b:branch state(γ, t') |= undesirable_branch(b)

GP8 Requirements Refinement Successfulness

At a certain point in time, all nonlocal requirements will be refined.

Formalisation:

∃t ∀n:nonlocalreq
state(γ, t) |= is_a_current_requirement(n) ⇒ state(γ, t) |= 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 1500.

Formalisation:

∀t state(γ, t) |= is_a_current_requirement(dodcheap) ⇒
∃t2>t ∃x:integer state(γ, t) |= total_DOD_costs(x) & x ≤ 1500

9.2 Checking Results

The properties have been checked against four different traces (i.e., those shown in Section 8 and Appendix C). As can be seen in this table, for 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 trace 3 and 4, since in these traces requirement revision takes place. Obviously, property GP5(1403) only holds for trace 4, since in the other traces the final DOD's have different costs.

Table 2. Results of Automated Checking

property	*trace 1*	*trace 2*	*trace 3*	*trace 4*
GP1	+	+	+	+
GP2	+	+	+	+
GP3	+	-	+	+
GP4	+	+	+	+
GP5(1403)	-	-	-	+
GP6	+	+	-	-
GP7	+	+	+	+
GP8	+	+	+	+
GP9	+	+	+	+

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 and 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 be automatically verified using techniques from (McMillan, 1993; Clarke et al., 1999; Sharpanskykh and Treur, 2006).

In combination with the automated checks of simulation traces described above, a hierarchy of interlevel relations can play an important role in the analysis of design processes, because of their hierarchical structure. In other words, if a certain Global Property turns out not to hold for a given design process 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. In this section, as an example some of these relationships between global properties and local properties of a design process are discussed. For the purposes of presentation they will only be described ion an informal/semiformal form.

One of the most relevant global properties of a design process is whether or not a set of requirements and a design object description are generated such that the design object description fulfils the set of requirements and both satisfy the stakeholder. However, without further assumptions this property is not guaranteed. Reasons why it may be hard to come to a result are, for example,

- the stakeholder may impose a too strong (inconsistent) set of requirements, 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 or not certain requirements or design object components are undesirable.
- the available set of components is too limited to fulfill the stakeholders requirements

In the first case the requirements can be inconsistent so that no design object exists that fulfils 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 design object description 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 behaviour, 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 design object descriptions 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:

$$\text{stabilizes}(\gamma, p, \text{pos}) \equiv \exists t\ \forall t' \geq t\ \text{state}(\gamma, t') \models p$$
$$\text{stabilizes}(\gamma, p, \text{neg}) \equiv \exists t\ \forall t' \geq t\ \text{state}(\gamma, t') \models \text{not}(p)$$
$$\text{stabilizes}(\gamma, p) \equiv \exists t\ [\ \forall t' \geq t\ \text{state}(\gamma, t') \models p \lor \forall t' \geq t\ \text{state}(\gamma, t') \models \text{not}(p)]$$

EP1

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

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

EP2

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

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

EP3

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

$$\exists \gamma\ \exists r_1, \ldots r_n\ [\ \forall r\ [\ \text{stabilizes}(\gamma, R(r), \text{pos}) \Leftrightarrow \textstyle\bigvee_i\ r = r_i\ ]\ \&$$
$$\forall r\ [\text{stabilizes}(\gamma, \text{undesirable_requirement}(r), \text{pos}) \Rightarrow \text{not stabilizes}(\gamma, R(r), \text{pos})\ ]\ \&$$
$$\forall r\ [\text{stabilizes}(\gamma, R(r), \text{pos}) \Rightarrow [\ \text{stabilizes}(\gamma, \text{undesirable_requirement}(r), \text{neg}) \lor$$
$$[\ \text{stabilizes}(\gamma, \exists r', b\ \ \text{is_a_subrequirement_of_via}(r, r', b) \land R(r'), \text{pos})\ \ \&\ \ \text{stabilizes}(\gamma, R(r'), \text{pos})\ ]\ ]\ \&$$
$$\forall r\ \ \text{stabilizes}(\gamma, R(r), \text{pos}) \Rightarrow [\ [\ \text{stabilizes}(\gamma, \exists q, b\ \ \text{is_a_subrequirement_of_via}(q, r, b) \land R(q), \text{pos})\ \ \&$$
$$\text{stabilizes}(\gamma, R(q), \text{pos})] \lor \exists c\ \text{stabilizes}(\gamma, \text{holds_for}(r, c), \text{pos})\ ]\ ]$$

EP4

At least one design object description exists that fulfils a set of fully refined requirements that does not contradict a stakeholder's stabilised (un)desirability preferences for requirements, and that does not contradict the stakeholder's stabilised (un)desirability preferences for components.
Formally:

$$
\begin{array}{l}
\exists\gamma\,\exists r_1, \ldots r_n\ [\ \forall r\ [\ \text{stabilizes}(\gamma, R(r), \text{pos}) \Leftrightarrow \bigvee_i\ r = r_i\ ]\ \& \\
\forall r\ [\text{stabilizes}(\gamma, \text{undesirable_requirement}(r), \text{pos})\ \Rightarrow\ \text{not stabilizes}(\gamma, R(r), \text{pos})\ ]\ \& \\
\forall r\ [\text{stabilizes}(\gamma, R(r), \text{pos})\ \Rightarrow\ [\ \text{stabilizes}(\gamma, \text{undesirable_requirement}(r), \text{neg})\ \vee \\
[\ \text{stabilizes}(\gamma, \exists\, r', b\ \text{is_a_subrequirement_of_via}(r, r', b) \wedge R(r'), \text{pos})\ \ \&\ \ \text{stabilizes}(\gamma, R(r'), \text{pos})\ ]\ ]\ \& \\
\forall r\ \ \text{stabilizes}(\gamma, R(r), \text{pos})\ \Rightarrow\ [\ [\ \text{stabilizes}(\gamma, \exists\, q, b\ \ \text{is_a_subrequirement_of_via}(q, r, b) \wedge R(q), \text{pos})\ \ \& \\
\text{stabilizes}(\gamma, R(q), \text{pos})]\ \vee\ \exists c\ \text{stabilizes}(\gamma, \text{holds_for}(r, c), \text{pos})\ ]\ \& \\
\exists d\ \text{stabilizes}(\gamma, \forall r\ R(r)\ \Rightarrow\ \exists c\ \text{part_of}(c, d)\ \&\ \text{holds_for}(r, c), \text{pos})\ ]
\end{array}
$$

Note that, also under such assumptions (leaving no doubt on the existence of a suitable requirement set and design object description), 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 design object description. Note, moreover, 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 design object description 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 stabilise. This property is related to the fact that the number of branches is finite, and that for a given (stabilised) set of undesirable components, for each branch a unique number is determined. These properties can be formalised in a manner similar as the ones above.

IPDOD1

If after some time the stakeholder's (un)desirability preferences for design object description components stabilise
and after some time a stabilised fully refined stable set of current requirements occurs
then after some time the branch costs stabilise

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

IPDOD2

If after some time a stabilised fully refined stable set of current requirements occurs
and after some time the stakeholder's (un)desirability preferences for design object description components stabilise
and at least one design object description exists that fulfils the stabilised set of fully refined requirements that does not contradict the stakeholder's stabilised (un)desirability preferences for components, with branch costs below the expected branch costs
then after some time a stabilised design object description occurs that fulfils the stabilised set of fully refined requirements that does not contradict the stakeholder's stabilised (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.

IPRQ1

If after some time the stakeholder's (un)desirability preferences for requirements stabilise
and after some time the stakeholder's (un)desirability preferences for design object description components stabilise
and after some time the branch costs stabilise
and at least one design object description exists that fulfills a set of fully refined requirements that does not contradict a stakeholder's stabilised (un)desirability preferences for requirements, and that does not contradict the stakeholder's stabilised (un)desirability preferences for components.
then after some time a stabilised fully refined stable set of current requirements occurs that does not contradict the stakeholder's stabilised (un)desirability preferences for requirements

The top level property considered is as follows:

GP0

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

The logical interlevel relationships are as follows:

EP1 & EP2 & EP3 & EP4 & IPDOD1 & IPDOD2 & IPRQ1 ⇒ GP0

In the graphical form of an AND-tree these interlevel relationships are depicted in Figure 5. Here EP stands for all (required) environmental properties. 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*.

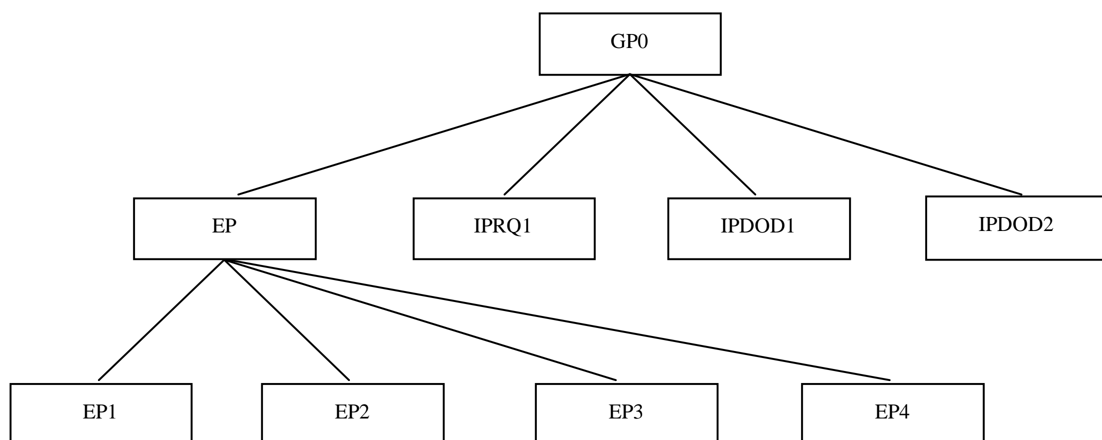

Figure 5 Logical interlevel relationships between dynamic properties of a design process

11. Discussion

In order to develop automated support for the dynamics of nontrivial design processes, the challenge of modelling and analysing such dynamics in a formal manner has to be addressed; cf. (Brown and Candrasekaran 1989; Corkill 2000; Heller and Westfechtel 2003; Baldwin and Chung 1995). This paper offers an approach to do so. The complex dynamics of a design process has been analysed 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 result of simulation has been checked against the properties of the design process as a whole.

Compared to the references mentioned above, the approach put forward is a declarative, logical approach supported by a formal language TTL for specification of dynamic properties of design processes, which has a high expressivity; cf. (Jonker and Treur, 2002). Furthermore, also simulation models are specified in a declarative, logical manner, which allows using these specifications in logical analysis as well; cf. (Bosse et al., 2005).

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.

The analysis approach that is for the first time applied to design processes here, has previously been applied to complex and dynamic reasoning processes other than design, such as reasoning by dynamically adding and evaluating assumptions (Bosse, Jonker and Treur 2006; Jonker and Treur 2003), and reasoning based on multiple representations (Bosse, Jonker and Treur 2003). In these cases in addition to simulated traces, also empirical (human) reasoning traces have been formally analysed. For further research it is planned to formally analyse protocols of human design processes in a similar manner, using methods as, for example, described in (Nagai and Taura, 2006).

References

Baldwin and Chung (1995). A Formal Approach to Managing Design Processes. *IEEE Computer*, Feb. 1995, pp. 54-63

Barringer, H., Fisher, M., Gabbay, D., Owens, R., and Reynolds, M. *The Imperative Future: Principles of Executable Temporal Logic*, John Wiley & Sons, 1996.

Bosse, T., Jonker, C. M., van der Meij, L., and Treur, J. (2005). LEADSTO: a Language and Environment for Analysis of Dynamics by SimulaTiOn. In: Eymann, T. et al. (eds.), *Proceedings of the Third German Conference on Multi-Agent System Technologies, MATES'05*. Lecture Notes in AI, vol. 3550. Springer Verlag, pp. 165-178.

Bosse, T., Jonker, C.M., and Treur, J., (2003). Simulation and analysis of controlled multi-representational reasoning processes. *Proc. of the Fifth International Conference on Cognitive Modelling, ICCM'03*. Universitats-Verlag Bamberg, 2003, pp. 27-32.

Bosse, T., Jonker, C.M., and Treur, J., (2006). Reasoning by Assumption: Formalisation and Analysis of Human Reasoning Traces. *Cognitive Science Journal*, vol. 20, 2006, pp. 147-180.

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

Brazier F.M.T., Langen P.H.G. van, Treur J., (1996). A logical theory of design. In: J.S. Gero (ed.), *Advances in Formal Design Methods for CAD, Proc. of the Second International Workshop on Formal Methods in Design*. Chapman & Hall, New York, 1996, pp. 243-266.

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

Clarke, E.M., O. Grumberg, D.A. Peled, (1999). *Model Checking*, MIT Press, Cambridge Massachusetts, London England.

Corkill, D.D. (2000). When Workflow Doesn' t Work: Issues in managing dynamic processes. *Proceedings of the Design Project Support using Process Models Workshop*, Sixth International Conference on Artificial Intelligence in Design, Worcester, Massachusetts, June 2000, pp. 1-13.

Fisher, M. (2005). Temporal Development Methods for Agent-Based Systems, *Journal of Autonomous Agents and Multi-Agent Systems*, vol. 10, pp. 41-66.

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

Galton, A. (2006). Operators vs Arguments: The Ins and Outs of Reification. *Synthese*, vol. 150, 2006, pp. 415-441.

Gavrila, I.S., and Treur, J. (1994). A formal model for the dynamics of compositional reasoning systems. In: A.G. Cohn (ed.), *Proc. of the 11th European Conference on Artificial Intelligence, ECAI'94*, Wiley and Sons, 1994, pp. 307-311

Gero, J., and Kannengiesser, U. (2006). A function-behaviour-structure ontology of processes. In: J.S. Gero (ed.), *Design Computing and Cognition '06, Proc. of the Second International Conference on Design Computing and Cognition, DCC'06*. Springer Verlag, 2006, pp. 407-422.

Goldblatt, R. *Logics of Time and Computation*, 2nd edition, CSLI Lecture Notes 7, 1992

Heller, M., and Westfechtel, B. (2003). Dynamic Project and Workflow Management for Design Processes in Chemical Engineering, *Proceedings 8th International Conference on Process Systems Engineering (PSE 2003)*, Kunming, China (June 2003)

Jonker, C.M., and Treur, J. (2002). Compositional Verification of Multi-Agent Systems: a Formal Analysis of Pro-activeness and Reactiveness. *International Journal of Cooperative Information Systems*, vol. 11, 2002, pp. 51-92. Preliminary version in: W.P. de Roever, H. Langmaack, A. Pnueli (eds.), *Proceedings of the International Workshop on Compositionality, COMPOS'97*. Lecture Notes in Computer Science, vol. 1536, Springer Verlag, 1998, pp. 350-380.

Jonker, C.M., and Treur, J., (2003). Modelling the Dynamics of Reasoning Processes: Reasoning by Assumption. *Cognitive Systems Research Journal,* vol. 4, 2003, pp. 119-136.

Jonker, C.M., Treur, J., and Wijngaards, W.C.A., (2002). Requirements Specification and Automated Evaluation of Dynamic Properties of a Component-Based Design. In: J. Gero (ed.), *Artificial Intelligence in Design '02, Proceedings of the Seventh International Conference on AI in Design, AID'02.* Kluwer Academic Publishers, 2002, pp. 547-570.

Jonker, C.M., Treur, J., and Wijngaards, W.C.A. (2003). A Temporal Modelling Environment for Internally Grounded Beliefs, Desires and Intentions. *Cognitive Systems Research Journal,* vol. 4(3), 2003, pp. 191-210.

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

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

McMillan, K.L., (1993). *Symbolic Model Checking: An Approach to the State Explosion Problem.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, 1992. Published by Kluwer Academic Publishers, 1993.

Nagai, Y., and Taura, T., (2006). Formal description of concept-synthesizing process for creative design. In: J.S. Gero (ed.), *Design Computing and Cognition '06, Proc. of the Second International Conference on Design Computing and Cognition, DCC'06*. Springer Verlag, 2006, pp. 443-460.

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

Sharpanskykh, A., and Treur, J., Verifying Interlevel Relations within Multi-Agent Systems. In: *Proc. of the 17th European Conference on Artificial Intelligence, ECAI'06.* IOS Press, to appear, 2006.

Treur, J., (1989). A logical analysis of design tasks for expert systems, *International Journal of Expert Systems*, vol. 2, 1989, pp. 233-253. See also: Treur, J., A logical framework for design processes. In: P.J.W. ten Hagen, P.J. Veerkamp (eds.), *Intelligent CAD Systems III. Proc. of the Third Eurographics Workshop on Intelligent CAD Systems*, Springer Verlag, 1991, pp. 3-20.

Appendix A: Dynamic Properties for the Case Study on a multi-agent system for information gathering

Global properties

DODGP The designed system is able to correctly classify any object of the given set

At the highest level, the overall property GP1 is refined making use of the following three properties.

DODGP1. For each type of information there is a component that initiates information acquisition.
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.
DODGP3. If E generates information of each type, then by some component within the system a conclusion will be generated.

Interaction properties

DODI1(X). If X initiated information acquisition, then E will provide the required information for X.
DODI2(X, Y). If E provides information of type IT for Y, then Y will communicate this information to X.
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 Z.

Behavioural properties for a component X

DODBP1(X). X is information acquisition pro-active
DODBP2(X). X is request pro-active
DODBP3(X). X is conclusion pro-active
DODBP4(X). X is information acquisition reactive
DODBP5(X). X is information provision reactive
DODBP6(X). X is information provision pro-active

Properties for component E

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

Transfer properties

DODTP(X, Y). Communication generated by X for Y is received by Y
DODTP(X, E). Initiated information acquisition by agent X is received by component E
DODTP(E, X). Information made available by E for X is received by X

Properties for branches

B1. For each type of information, there is a component that pro-actively initiates information acquisition.
B2. For information of type IT1 there is a component that pro-actively initiates information acquisition, and for information of the different type IT2 there is a component that pro -actively requests the information and another component that reactively initiates information acquisition.
B3. For each type of information, there is a component that pro-actively requests the information and another component that reactively initiates information acquisition.
B4. If E provides information of type IT for Y, then Y will pro -actively provide this information for X.
B5. If E provides information of type IT for Y, then Y will reactively provide this information for X.

Appendix B: Leadsto Specification

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), …}

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)

Local Properties

Requirements

LP0 Initialisation

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

start $\rightarrow$ is_a_current_requirement(dodgp) $\wedge$ is_a_current_requirement(dodcheap)

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.
Formalisation:

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)

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).
Formalisation:

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

LP2 Undesirable Branch Determination

These local properties are used to determine which branches are undesirable. There are two cases: (1) a requirement that belongs to it is undesirable and (2) its total costs are higher than predicted, whilst the requirement dodcheap is present.
Formalisation:

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)

LP3 Branch Selection

These five local properties are used to determine which is the best branch (or subtree) that can be selected in order 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.
Formalisation:

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)

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).
Formalisation:

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)

Design Object Description

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.
Formalisation:

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)

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).

Formalisation:

‘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)

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.

Formalisation:

current_DOD(dod(x)) ∧ ‘DOD_counter’(x) → current_DOD(dod(x))

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.

Formalisation:

part_of_DOD(c,d) → part_of_DOD(c,d)

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.

Formalisation:

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

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, (1) it is not a current branch OR (2) it is not a "leaf" branch OR (3) its costs do not exceed the predicted costs.

Formalisation:

‘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)

LP10 Total Branch Costs Calculation

These properties are used to calculate the total costs for a certain branch. In order 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.

Formalisation:

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,l,x) ∧ not_index(b,l,y) → not_index(b,l+1,x)
intermediate_branch_costs(b,x) ∧ intermediate_branch_costs(b,y) ∧ x<y ∧ not_index(b,l,y) ∧ not(not_index(b,l,x)) ∧
not(hasindex(b,x)) ∧ sum(b,l,s) → index(b,l+1,x) ∧ hasindex(b,x) ∧ hassum(b,l+1) ∧ sum(b,l+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)

LP11 Total DOD Costs Calculation

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

Formalisation:

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(l,x) ∧ not_index(l,y) → not_index(l+1,x)
intermediate_DOD_costs(x) ∧ intermediate_DOD_costs(y) ∧ x<y ∧ not_index(l,y) ∧ not(not_index(l,x)) ∧ not(hasindex(x)) ∧ sum(l,s)
→ index(l+1,x) ∧ hasindex(x) ∧ hassum(l+1) ∧ sum(l+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)

P12 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.
Formalisation:

is_a_current_requirement(n) ∧ not(requirement_refined(n)) → ' RQSM_active'

LP13 DODM Activity Determination

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

' RQSM_active' → ' RQSM_earlier_active'
' RQSM_earlier_active' → ' RQSM_earlier_active'
' RQSM_earlier_active' ∧ not(' RQSM_active') → ' DODM_active'

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.
Formalisation:

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)

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 (1) when the requirement is not required for the given design object, (2) when the requirement has been refined, and (3) when the requirement is satisfied.
Formalisation:

' 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)

LP16 Termination Determination

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

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

Appendix C : Additional Simulation Traces

Trace 1 & Trace2 : see Section 8

Trace 3