Feature Interaction Detection in Building Control Systems by Means of a Formal Product Model

Andreas METZGER, Christian WEBEL
Department of Computer Science, University of Kaiserslautern
P.O. Box 3049, 67653 Kaiserslautern, Germany
{metzger, webel}@informatik.uni-kl.de

Abstract. The complexity of present-day software systems has reached dimensions that require systematic approaches for coping with it. In addition to traditional domains, such as telecommunications or business applications, complex software systems can be found in the domain of reactive systems, of which building control systems are an interesting example. The extension and reuse of these systems have become important activities in the respective development processes. To be able to correctly execute these activities, the developers need to be aware of interactions that might exist between different features of a system.

In this paper, an approach for the systematic detection of feature interactions in building control systems is presented, which allows the automatic identification of such interactions based on existing requirements specification documents. This is realized by the application of a formal model of the development products, which includes traceability relations between these products.

Introduction[1]

The complexity of modern reactive systems is on a steady rise. An interesting example of this is integrated building control systems, which are essential for optimizing the total building performance [1]. To provide optimal results, such systems have to consider the interaction of different physical effects, which most often requires a strong coupling of different parts of such control systems. Further, constructing building control systems expose a huge complexity due to the sheer number of objects that needs to be dealt with; e.g., the integrated control system that was developed for a floor of our university building consists of 920 objects [2].

Because of this complexity, a number of problems arise. One of these problems is the extension of such systems with additional functionality. This is typically required if new user needs should be considered. Besides the desired behavior, this new functionality might introduce *undesirable* interrelationships with old parts of the system, and thus an unwanted system behavior might be the result. To illustrate this, we assume an exemplary building control system that allows the automatic control of the illumination inside a room. In this system, the cost of artificial lighting is reduced by employing daylight whenever possible. If this system is extended to satisfy the user need of avoiding glare at the workplace, an undesirable interrelation might be introduced, such that daylight might be used in situations that produce glare.

Another problem can be discovered when complex reactive systems should be reused. Simply eliminating the parts that are not needed in the new context can result in faulty systems, because *required* interrelations to the parts that have been eliminated might not have

1. This paper has been published in D. Amyot, L. Logrippo (Eds.) *Feature Interactions in Telecommunications and Software Systems VII.* Amsterdam: IOS Press. 2003. pp. 105–121

been considered. To give an example for such a condition, we assume that the building control system above additionally realizes an alarm system. One functionality of this alarm system is to turn on all lights when an alarm has been triggered. If this system should be reused as an alarm system only, all parts that are relevant for lighting (lights, blinds, etc.) will probably be removed. However, if this removal is exercised without considering the required interrelations, a malfunctioning system is the result as the lights are also needed for the alarm system.

Although the above examples appear to be trivial, in real control systems, which typically realize between 200 [2] and 8000 [3] requirements, this detection cannot reasonably be performed manually. Therefore, we propose an approach for the *automatic* detection of such interactions. Our detection concept is based on a formal *product model*, which reflects each type of development product as well as its relations with other types of products. These relations are an important means for establishing traceability between products.

For the purpose of interaction detection, we employ a product model that is available as part of a precise definition of the requirements specification method *PROBAnD* [2]. After this product model has been instantiated using the set of available requirements documents, interaction information can be automatically derived from model data.

Related Work

In our opinion, the above problems are similar to the feature interaction problem, which has received considerable attention in the telecommunications domain [4]. Consequently, the product model-based approach presented in this paper might not only provide solutions for the depicted problems in the building control context, but could also prove to be valuable in the domain of telecommunication systems.

Several feature interaction detection approaches have been presented in the literature. In [5] an approach that detects undesirable interactions by employing Use Case Maps [6] and LOTOS [7] is introduced. After features have been semi-formally described with Use Case Maps, these features are formally defined with LOTOS, which then can be used for verification. Other authors also propose using formal description techniques for the purpose of interaction detection. In [8] and [9] temporal logic and model checking are suggested, in [10] a special heuristic is chosen, and in [11] as well as in [12] description logic and logic meta-programming are applied.

All of the above approaches require the modeler to explicitly and formally specify features. In contrast to this, our concept does not require the requirements engineer to provide any *additional* information during the specification process. The existing documents can be used exactly as they have been described in our requirements specification method *without* considering feature interaction. Therefore, this approach can provide an increase in product quality with only little additional effort.

However, this simplification does prevent us from distinguishing between interactions that are undesirable or required. Therefore, this has to be determined by the developers on the basis of the automatically analyzed interaction information and the existing documents. If an automation of this activity should be achieved, an extension of our requirements specification method would become necessary such that required interactions can be specified.

Where many approaches like [8] or [13] use actual implementation code or extended implementation code (e.g., meta-programming [11]) for detecting feature interaction, we are able to detect such interactions very early in the development process, i.e., during requirements specification. This has the major benefit of eliminating undesirable interactions before they find their way into the final product, where removing these interactions can become very costly.

In the remainder of this paper, the requirements specification method and its product model are described. Then, the concepts for applying this product model for the purpose of interaction detection are presented, which are illustrated by an example of a small building control system. Finally, the results of two complex case studies are discussed.

1. The Product Model

When defining a detailed product model that reflects all necessary development products, the model itself might become very complex. In our case, the complete model currently contains 80 different entities. Therefore, we structure our model by classifying the different types of entities according to Fig. 1.

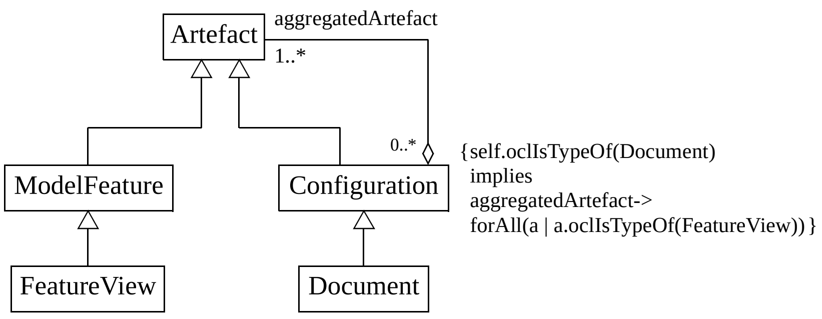

Figure 1. Classification of Development Products

The most general type of entity is called *artefact*, which can be identified with any development product. Each artefact has a unique name and can optionally be described by informal text.

More specialized types of entities are named *model-features*, which represent *atomic* development products. Note, that the notion of a model-feature is different from the general meaning of a feature, which is a self-contained functional part or aspect of a specification or system [14]. We will consistently be using the term model-feature in the remainder of this paper to make this distinction clear.

Model-features contain the actual development information independent of its representation. Thus, for each model-feature different *feature-views* exist, which contain model-feature information in a concrete representation, e.g. in HTML or SDL [15]. This is why feature-views can be conceived as specializations of model-features.

Additionally, *configurations* are introduced, which aggregate less complex artefacts. This is reflected by the aggregation relation in Fig. 1.

Usually, developers will be working on *documents*, which are composed of feature-views. Therefore, documents are classified as special configurations that are only allowed to aggregate feature-views. As documents inherit the aggregation relation of configurations, the constraint in Fig. 1 states that only feature-view instances can be at the *aggregatedArtefact* end of the relation.

This classification of product model entities further provides a means for reducing the number of artefacts that need to be handled for interaction detection. As model-features contain the same information as feature-views do, the detection concept can work on the level of model-features without neglecting any information. For that purpose, the set of available development documents is used for instantiating the model-features as well as the relations between these model-features. This step is carried out by decomposing the documents into feature-views and extracting model-feature information from these feature-views. Details on how this is performed can be found in [16].

With these prerequisites, our interaction detection approach is independent of the notation of the specification documents. As a consequence, the model-features of the product model, which will be described in Sect. 1.2, suffice as input for feature interaction detection.

1.1 The PROBAnD Method

Before the concrete model-features are depicted, an introduction to the PROBAnD requirements specification method [2][17] seems appropriate.

In Fig. 2, an overview of the documents and activities of the PROBAnD method is provided. The method takes the *problem description* as an input, which is divided into the *building description* and a collection of *needs*.

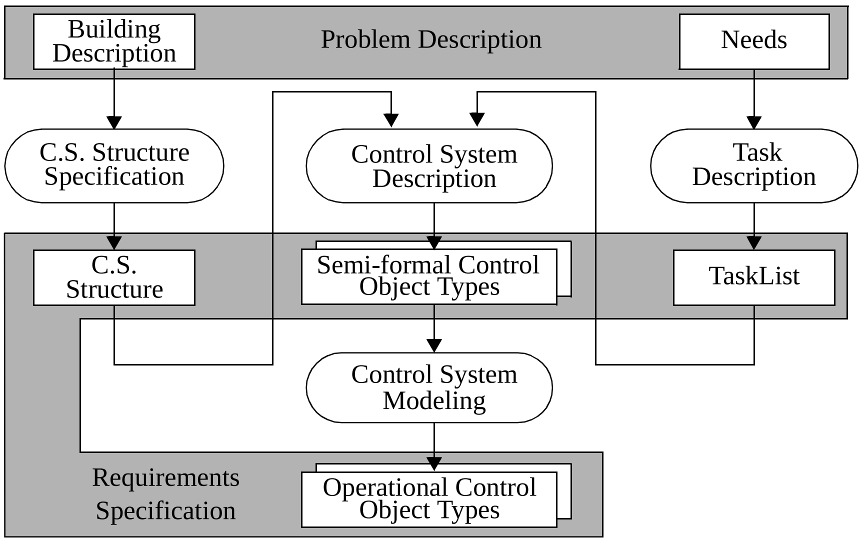
Building Description
Problem Description
Needs
C.S. Structure Specification
Control System Description
Task Description
C.S. Structure
Semi-formal Control Object Types
TaskList
Control System Modeling
Requirements Specification
Operational Control Object Types

Figure 2. Overview of the PROBAnD Method

The building description contains a description of the building's structure; e.g., a floorplan that shows that the building is made up of one floor with three rooms. Further, the building's installation is depicted; e.g., in an informal text that states: “There are one light, one illumination sensor, and one motion-detector in each room”.

From this building description, the *control system structure* is derived by considering the structure of the building. In the building automation domain this has proved to be a suitable approach as *control objects* (the objects of the control system) are most often identical to the building's objects or are a subset of these; e.g., lighting needs to be controlled only within the boundaries of one room, and therefore the control object for lighting can be derived from the room object. Additionally, sensors and actuators represent the interface of the control system to its environment and as such should be reflected in the specification.

To handle the huge number of control objects that need to be considered for large building control systems, *control object types* are formed, which are aggregated according to the hierarchy of the building's objects.

The other part of the problem description is made up by a collection of needs (i.e., user requirements). These needs informally describe the control system from the point of view of the users in natural language.

These needs have to be split into less complex *tasks* (i.e., developer requirements) such that they can be assigned to a single control object type.

After the control object types have been identified and tasks have been assigned, *strategies* for realizing the control tasks are informally given in natural language, leading to a collection of *semi-formal control object types*. From these control object types, *operational control object types* are specified in SDL, from which control system prototypes can be generated.

It should be noted that control object types are introduced during the requirements specification phase solely for the purpose of structuring the specification in a suitable way. In the design phase, this structure can be re-arranged. Furthermore, the specification of strategies should be understood as an exemplary solution of the developer requirements. This is needed for generating prototypes, which are an important means for the early validation of user requirements [18]. During the design phase, different solutions for the strategies are possible and legitimate.

1.2 The Product Model of PROBAnD

As it has been introduced in the previous section, an input to the *PROBAnD* requirements engineering method are the user requirements, called *needs*, which are shown in Fig. 3 as a specialization of the model-feature *requirement*. Only needs that represent functional requirements, i. e., *functional needs*, are regarded during our requirements specification process. Because of their granularity, we interpret these functional needs as features according to [14].

Needs are realized by *tasks*, which themselves might be realized by other tasks. This dependency is modelled by the *realizedBy* relation between *requirement* and *task* in Fig. 3. This relation—like all other relations of the product model—are explicitly reflected in the development documents, which are used for instantiating the model-features.

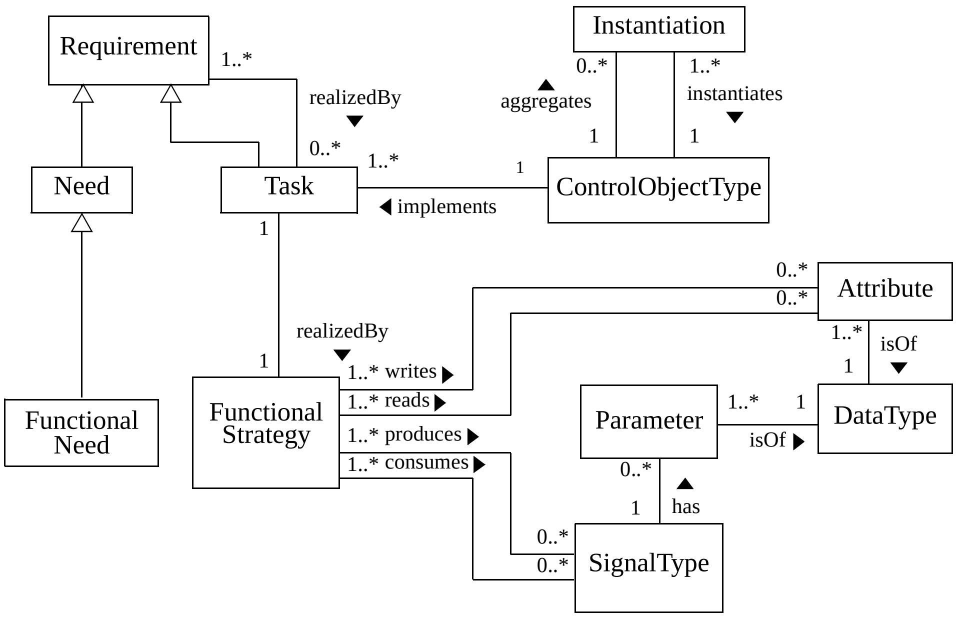

Figure 3. Types of Model-Features

Each task describes a responsibility that has to be fulfilled by the one *control object type* that implements this very task. This is reflected by the one-to-many *implements* relation from the model-feature *control object type* to the model-feature *task*. As it has been depicted in Sect. 1.1, these control object types are always instantiated in a strict aggregation hierarchy, which leads to a tree of instances. The strict aggregation is modelled by the one-to-many *aggregates* relation between the model-feature *control object type* and the model-feature *instantiation*.

For each task, a *functional strategy* is specified. As it has been pointed out, such a strategy describes a possible solution for realizing the responsibility of the task. To establish communication between strategies of different tasks, strategies can read and write *attributes* as well as produce and consume signals that are of globally defined *signal types*, which can possess *parameters*. Where attributes are used for the communication between tasks of the *same* control object type, signals are used for exchanging data between tasks of *different* object types. It is important to note that signals are only allowed to travel along the aggregation hierarchy. This for example implies, that if control objects at the same level of the hierarchy need to communicate, the signals have to be routed through the parent instance, which aggregates the communicating instances.

To illustrate the model-features of the product model and to provide an example for the remainder of this paper, a small building control system is presented in the following subsection.

1.3 Small Building Control System Example

Three needs of our small control system example have already been described in the introduction. The first need ($N1$) can be expressed as "Provide required illumination in a room if it is occupied". The second need ($N2$) is "Use daylight to reduce energy consumption", and the third need ($N3$) is "Avoid glare at the workplace". To make this system a little more complex, a further need is added. This need ($N4$) states "Provide required temperature in a room". Table 1 lists these needs.

Table 1. Needs for a Small Building Control System

Need	Description
N1	Provide required illumination in a room if it is occupied.
N2	Use daylight to reduce energy consumption.
N3	Avoid glare at the workplace.
N4	Provide required temperature in a room.

According to the *PROBAnD* method, these needs have to be refined, leading to a collection of tasks, which are assigned to specific control object types. A possible structure for the small building control system derived from the respective building description is presented in Fig. 4.

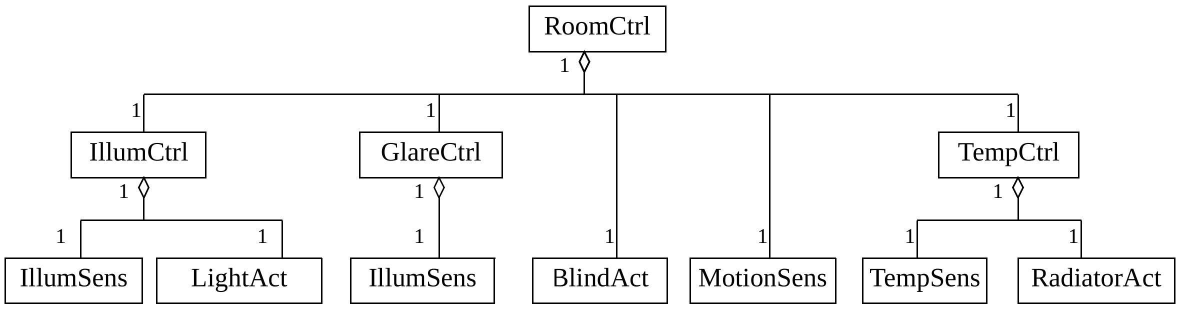

Figure 4. Structure of a Small Building Control System

The sensors and actuators, which represent the interface to the environment, can typically be found as leaf nodes in such a structure; e.g., *IllumSens* (IlluminationSensor) or *BlindAct* (BlindActuator) in Fig. 4.

Table 2 lists suitable tasks for refining the above needs. One such refinement for need $N3$ is provided by tasks $T6$ and $T3$, which describe the developer requirement of avoiding glare by employing the blinds. For this example, it is assumed that the building description provides blinds as the only means of shading.

The complete set of development documents of this small example can be found online at [19].

Table 2. Task List of a Building Control System

Task	Description	realizes	implementedBy
T1	If room is occupied, control indoor illumination with available light sources (blind and light), taking energy consumption into account.	N1, N2	IllumCtrl
T2	Turn light on or off on request.	T1	LightAct
T3	Open or close blind on request.	T1, T6	BlindAct
T4	Determine and report motion.	T1, T6	MotionSens
T5	Determine and report current illumination.	T1, T6	IllumSens
T6	Avoid glare at the workplace by using the blind if room is occupied.	N3	GlareCtrl
T7	Control room temperature by using the radiator.	N4	TempCtrl
T8	Open or close radiator valve on request.	T7	RadiatorAct
T9	Determine and report current temperature.	T7	TempSens

2. Feature Interaction Detection

In Sect. 1 we have outlined that functional needs can be interpreted as features. So, the goal of detecting feature interactions in building control systems can be reached by identifying dependencies between functional needs. These dependencies can be extracted by following the relations between model-features in an instantiation of the product model. Depending on the available level of information, results with different precision can be attained. In the following subsections, we will present how four different levels of information can be used for feature interaction detection.

2.1 Detection at Requirements Level

Early in the requirements specification process, needs and tasks are the only model-features that will have been specified. Therefore, interactions at this level can only be identified by employing these model-features together with the *realizedBy* relation between them. From this type of relation, a dependency graph between requirements (i. e., needs and tasks) can be attained. Fig. 5 shows such a graph, which depicts the dependencies between the requirements of the building control example. In such a graph, *points of interaction* can be identified, which are nodes that have more than one direct parent.

According to Table 2, the needs $N1$ and $N2$ are realized by task $T1$, where $T1$ is realized by $T2$ to $T5$. Additionally, tasks $T3$ to $T5$ are needed for fulfilling need $N3$. This leads to four points of interaction that can be identified: $T1$, $T3$, $T4$, and $T5$.

To formalize how such points of interaction can be found, a few basic definitions will be provided in the following paragraphs.

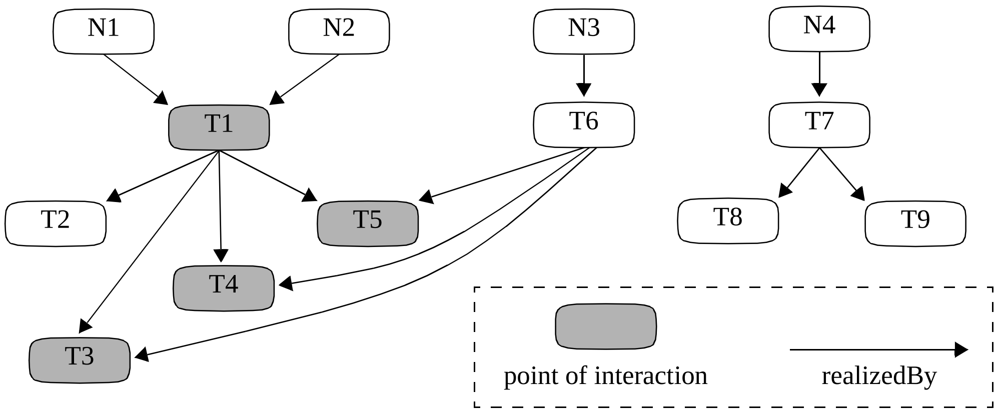

Figure 5. Dependency Graph for Requirements

Let $N = \{N_1, \ldots, N_n\}$ be the set of all needs and $T = \{T_1, \ldots, T_m\}$ be the set of all tasks. As both needs and tasks are requirements, the set of all requirements consequently is defined by $R = \{N_1, \ldots, N_n, T_1, \ldots, T_m\} = \{R_1, \ldots, R_{n+m}\}$. Further, we define a *realizedBy* predicate such that

$R_i \rightarrow R_j$ **iff** requirement R_i is realized by requirement R_j.

Obviously, this predicate is transitive; e.g., $N1 \rightarrow T4$ holds in Fig. 5. With these definitions, we can define a predicate φ for the point of interaction with

$\varphi(T_i)$ **iff** there exists a set $N(T_i) = \{N_{i,1}, \ldots, N_{i,k}\} \subseteq N$ with $|N(T_i)| > 1$ such that $N_j \rightarrow T_i$ for each $N_{i,j} \in N(T_i)$.

Accordingly, in our building control system $\varphi(T1)$, $\varphi(T3)$, $\varphi(T4)$, $\varphi(T5)$ hold.

To algorithmically determine these points of interaction, we initialize $N(T_i)$ with the empty set for $i = 1, \ldots, m$. Then, for each need $N_t \in N$ ($t = 1, \ldots, n$), we follow all *realizedBy* relations and add N_t to the set $N(T_i)$ for each task T_i that is traversed. As a result, each task T_i will have been marked by a set $N(T_i)$, from which $\varphi(T_i)$ can be computed according to the above formula.

Because our approach only employs needs and tasks at this level, it can be conceived as being domain and method independent. Especially, this implies that whenever a traceability relation from features (i.e., needs) to responsibilities (i.e., tasks) exists in a given specification method or language, the above algorithm can be applied. An example for such a language is the User Requirements Notation (URN [20]), which provides the necessary means for requirements traceability.

With the knowledge of the points of interaction, we are only able to systematically determine *possible* feature interactions, because dependencies at the level of needs and tasks do not necessarily imply that there will be interactions in the final system. For example, the task at a given point of interaction might never be used for realizing more than one need simultaneously. Therefore, from the points of interaction of Fig. 5, only a possible feature interaction between the needs $N1$ and $N2$ (at $T1$) as well as a possible interaction between $N1$, $N2$, and $N3$ (at $T3$, $T4$, and $T5$) can be derived.

2.2 Detection at Strategy Level

In order to refine this set of possible feature interactions, dependencies between tasks by means of their strategies can be examined. This dependency occurs because strategies are coupled by signal types or attributes (see Fig. 3). Two observations can be made when using this information.

The first observation is that, in spite of a possible interaction detected on the level of the requirements, an interaction between two tasks cannot occur if these tasks only consume (resp. read) signals (resp. attributes) that are produced (resp. written) by the task at the point of interaction. An example for this is task $T4$ of our small control system. As this motion sensor only produces signals, which are consumed (by $T1$ and $T6$), there will be no interaction.

The detection of such situations can be carried out easily by following the *produces/consumes* (resp. *writes/reads*) relations from the model-feature *functional strategy* to the model-feature *signal type* (resp. *attribute*).

It should be mentioned that we are working on requirements specification documents, only. Therefore, problems that could arise in the actual implementation of the system are not considered. One of these problems could be that the communication between objects is not reliable, which would lead to an unwanted interaction if the information from the motion sensor could not be received.

The second observation that can be made when employing strategy information is that, with the introduction of strategies, additional interactions can originate. It is possible for the developer to have the strategies of two independent requirements operate on the same set of data (e.g., by using the same attribute), thus introducing a coupling between the tasks. In our small example, this situation does not occur.

To elaborate this point, depending on the chosen realization of a strategy, the number and types of interactions can vary. This especially implies that when the development process enters the design phase, some of the interactions that have been identified during the requirements specification phase need to be re-evaluated.

2.3 Detection at Object Structure Level

After the above levels of information have been considered, the list of possible interactions is already refined. In order to further improve the set of possible feature interactions, interactions that cannot occur should be eliminated.

A step towards this goal can be taken by considering the aggregation hierarchy of the control object types. This information provides clues as to whether an interaction is possible, because control object types, which implement the respective tasks, might be instantiated in such a way that dependent tasks will never be used at the same point of instantiation. An example for this can be given for task $T5$, which is implemented by the control object type $IllumSens$. The depending tasks $T1$ and $T6$ are implemented by different control object types ($IllumCtrl$ and $GlareCtrl$ respectively). Because both individually aggregate an instance of $IllumCtrl$ there will be no interaction between tasks $T1$ and $T6$ at the point of $T5$.

To formalize this fact, let $C = \{C_1, \ldots, C_l\}$ be the set of all control object types and $C(T_i)$ the control object type that implements task T_i. Further, $P(C_k, C_l)$ should be the set of all control object types that are instantiated on the *shortest paths* from the control object type C_k to C_l. For instance, in the small building control example

$$P(IllumSens, GlareCtrl) = \{IllumSens, GlareCtrl\}.$$

For evaluating $P(C_k, C_l)$, the *aggregates* and *instantiates* relations of the product model are considered. The calculation of the shortest paths starts at C_k, from which all control object types that can be found along the aggregation hierarchy to the point(s) where C_l is instantiated are determined. Typically, this requires passing through the hierarchy towards the root object type using the reverse direction of the *instantiates* and the *aggregates* relation. Once the root is reached, the tree is searched in a downwards fashion using the forward direction of the *aggregates* and *instantiates* relation. Of all possible paths that have been determined with this algorithm, the shortest ones are calculated and the all object types along these shortest paths form the required set $P(C_k, C_l)$.

To illustrate, Fig. 6 shows an excerpt of an instantiation of the product model for the above example.

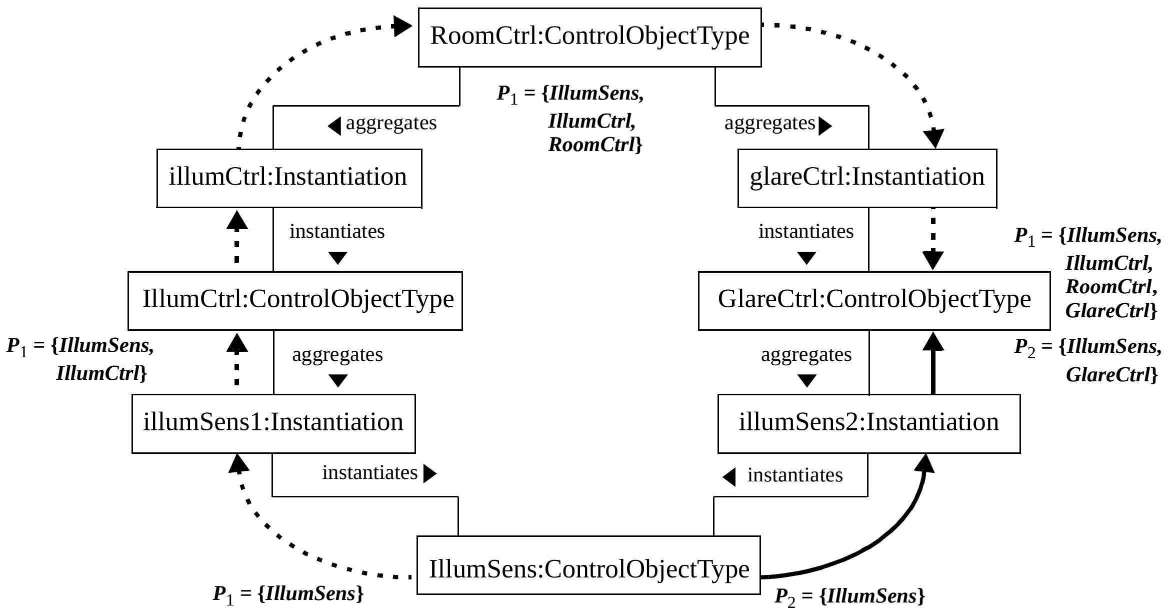
RoomCtrl:ControlObjectType
P1 = {IllumSens, IllumCtrl, RoomCtrl}
aggregates
aggregates
illumCtrl:Instantiation
glareCtrl:Instantiation
instantiates
instantiates
P1 = {IllumSens, IllumCtrl, RoomCtrl, GlareCtrl}
IllumCtrl:ControlObjectType
GlareCtrl:ControlObjectType
P1 = {IllumSens, IllumCtrl}
aggregates
aggregates
P2 = {IllumSens, GlareCtrl}
illumSens1:Instantiation
illumSens2:Instantiation
instantiates
instantiates
IllumSens:ControlObjectType
P1 = {IllumSens}
P2 = {IllumSens}

Figure 6. Calculation of Paths in an Instantiation of the Product Model

The calculation of the paths starts at the control object type $IllumSens$. Because two instantiations of this control object type exist, two possible paths (P_1 and P_2) can be identified. As P_2 is the shorter one, it can be deduced that

$$P(IllumSens, GlareCtrl) = P_2 = \{IllumSens, GlareCtrl\}.$$

With the technique for determining such paths, we can now eliminate the interactions that cannot occur because of the aggregation hierarchy.

A modeling guideline of the PROBAnD method requires that control object types that implement tasks that are connected by *realizedBy* relations should be instantiated as closely as possible, because this reduces the number of signals that need to be routed through other instances in the aggregation hierarchy. Therefore, only tasks that are directly and *not* transitively realized by the task at the point of interaction need to be considered.

If we assume that $T_d(T_i) = \{T_{d,1}, \ldots, T_{d,r}\}$ is the set of all tasks that are directly realized by T_i. The set of all control object types that are on the path between the object type that implement tasks T_i and the one that implements $T_{d,k}$ can be attainted by

$$P_{d,k} = P(C(T_i), C(T_{d,k})).$$

If $P_{d,s} \cap P_{d,l} = \{C(T_i)\}$, then there can be no interaction between tasks $T_{d,s}$ and $T_{d,l}$, because there are no common object types along the paths and as such different instantiations of the object type that implements the conflicting task are used.

If the intersections of all possible combinations of $P_{d,k}$ are computed, all interactions that cannot occur because of the instantiation hierarchy can be eliminated.

For the above example of task $T5$, $T_d(T5)$ is $\{T1, T6\}$, and thus

$$P_{d,1} = \{IllumSens, IllumCtrl\} \text{ and } P_{d,2} = \{IllumSens, GlareCtrl\},$$

which leads to

$$P_{d,1} \cap P_{d,2} = \{IllumSens\} = \{C(T5)\}.$$

Hence, there is no interaction at point $T5$. Whereas in the case of $T3$ the following sets are computed:

$$P_{d,1} = \{BlindAct, RoomCtrl, IllumCtrl\} \text{ and } P_{d,2} = \{BlindAct, RoomCtrl, GlareCtrl\},$$

which leads to

$$P_{d,1} \cap P_{d,2} = \{BlindAct, RoomCtrl\} \neq \{C(T3)\},$$

and therefore this interaction cannot be eliminated from the list of possible interactions. The same is valid for the point of interaction at $T4$.

Our approach at this level is strongly domain and method specific, because the static nature and the strict aggregation hierarchy of the building structure are important prerequisites for the detection algorithm. This implies that the application of this reduction concept is possible only for closely related domains, which expose similar properties; e. g., automotive control systems.

If all considerations that have been presented in Sect. 2.1 to Sect. 2.2 are exercised for our small building control system, the refined set of interactions that can be deduced contains the interaction of $\{N1, N2\}$ at $T1$ and the interaction of $\{N1, N2, N3\}$ at $T3$, which—at a closer look—can be identified as real feature interactions.

2.4 Detection at Environment Level

As our approach is targeted towards reactive systems, the environment of such systems has to be considered. This is especially important for the detection of feature interactions, because the physical environment can be the source of an implicit coupling between different parts of the control system.

This fact can already be discovered in our small building control example. The dependency graph in Fig. 5 shows no dependencies between need $N4$ (temperature control) and the other needs (lighting control). However, there exists a physical link between the room temperature and the amount of daylight that comes into the room. The reason for that is that sunlight can considerably heat up a space. Therefore, an interaction between $N4$ and the other needs should be noted.

To systematically determine such implicit interactions, the links in the physical environment must be made explicit for the detection process.

During the development of each reactive system, a simulator of the system's environment is needed for testing the dynamic behavior of the system. As such a simulator has to consider the physical links, the simulator's models can be used for making the links explicit, thus making this information available for the detection process.

In our case, building performance simulators for testing control systems have been modelled using the PROBAnD method [21][22]. By merging the product model instantiation for the control system and the product model instantiation for the simulation, we achieve a combined model instance. This combined model instance can be employed for uncovering feature interactions by applying the above algorithms.

To merge the two product model instances, *connection points* must be identified. As it was already pointed out, control object types that represent sensors and actuators are the interface to the environment. These control object types can now be used for identifying their counterparts in the simulator, thus establishing the required connection points.

This solution is sketched in Fig. 7 for our small control system. We already know that $T3$ is realized by the control object type $BlindAct$ and that $T9$ is realized by $TempSens$. If we assume that the task for simulating the blind actuator is named Ta and that the task for simulating the temperature sensor is named Tb, and if we further assume that both Ta and Tb are realized by task Tc, this very task can be identified as a point of interaction. This leads us to the conclusion that $N1$, $N2$ and $N3$ expose an interaction with $N4$ through the physical environment.

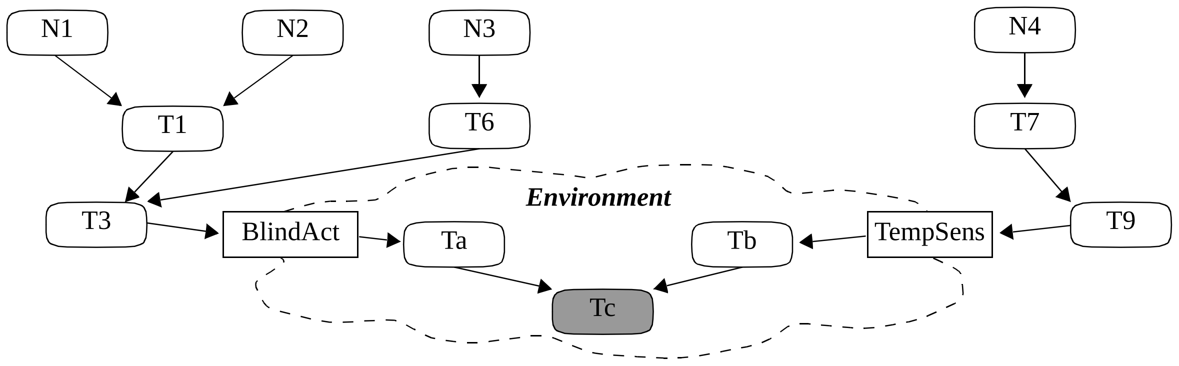

Figure 7. Dependency Introduced by Physical Environment

3. Results

To evaluate the above concepts and examine their feasibility in a real development context, tool prototypes have been implemented and applied in two case studies for different versions of a large building control system. The results of these case studies and a closer examination of the code as well as the run-time complexities of our tool will be provided in the following sections.

3.1 Case Studies

In the first case study, a heating and lighting control system, which we call *Floor32* in the remainder of this paper, was used as an example. A detailed description of this system and an analysis of qualitative and quantitative development data can be found in [23]. To give an impression of this system, a few of its 67 needs for heating and illumination are provided in Table 3.

Table 3. Needs for Large Heating and Illumination Control System (Excerpt)

Domain	Need	Description
Illumination	U2	As long as the room is occupied the chosen light scene has to be maintained.
	U3	If the room is reoccupied within t_1 minutes, the last chosen light scene has to be re-established
	U4	If the room is reoccupied after t_1 minutes, the default light scene has to be set.
	FM1	Use daylight to achieve the desired illumination whenever possible.
	FM6	The facility manager can turn off any light in a room or hallway section.
Heating	UH2	The comfort temperature shall be reached as fast as possible during heating up and maintained as best as possible afterwards.
	UH6	The user can manually move each sun blind up or down. This manual override holds until he/she leaves the room for a longer time period.
	FMH2	The use of solar radiation for heating should be preferred against using the central heating unit.

An extension of this system was used as a second example, which we call *Floor32X*. The extension of this system has been obtained by adding the functionality of an alarm system [24]. This is reflected by 12 additional requirements, some of which are listed in Table 4.

Table 4. Needs for Large Alarm System (Excerpt)

Domain	Need	Description
Alarm	UA2	If a person occupies a room with an activated alarm system, he/she can deactivate the alarm system by identifying himself/herself within t_2 seconds. Otherwise, the alarm must be triggered.
	UA3	If a room is unoccupied out of working hours for more than t_3 minutes, the alarm system must be activated automatically.
	UA10	If an alarm is triggered, all lamps in the corresponding sections are turned on. When the alarm is interrupted, the lamps are reset to their previous state.
	FA1	The facility manager can switch off an alarm, deactivate an alarm system, and activate the alarm system for an individual room or for all rooms of the building.

One might observe that these exemplary needs are very fine-grained and almost seem to describe the solutions rather than the actual requirements. This can be attributed to the fact that the persons that were in the role of the “customers” when specifying this system, were domain-experts, which are better at expressing solutions rather than actual problems.

To illustrate the structural complexity of *Floor32*, the simplified hierarchy of its 37 control object types is shown in Fig. 8. These control object types are instantiated to 920 control objects. when the system is executing.

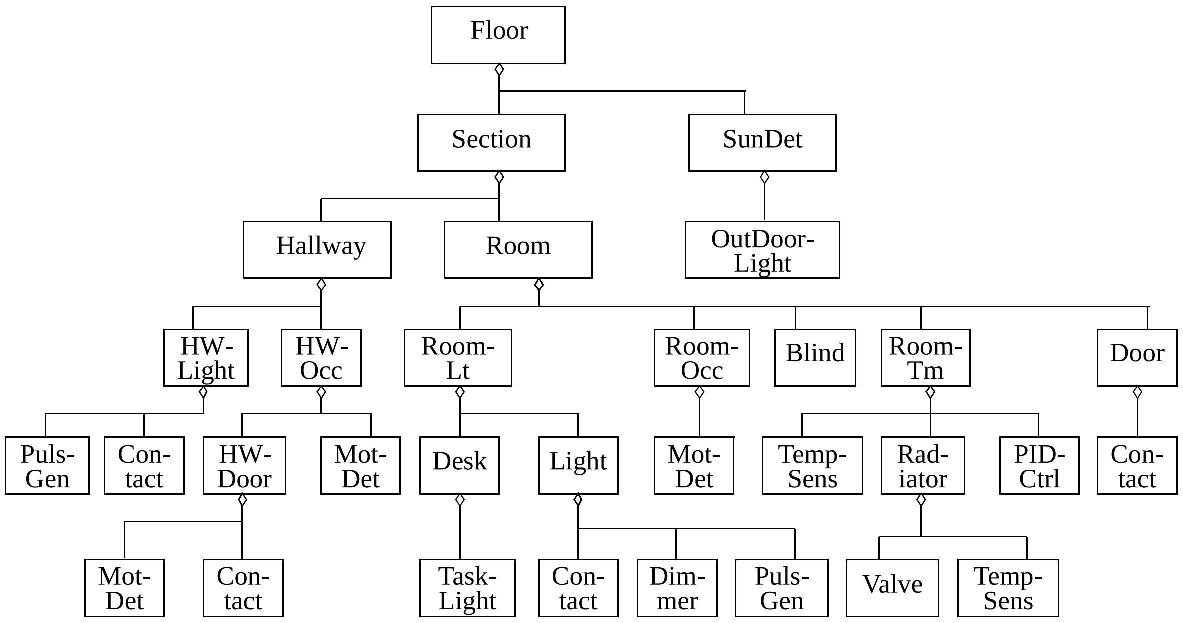

Figure 8. Object Structure of Complex Building Control System (*Floor32*)

To summarize, in *Floor32X*, a total number of 316 functional requirements (64 needs and 252 tasks) are realized, where in *Floor32* only 285 functional requirements (52 needs and 233 tasks) are implemented.

3.2 Detected Interactions

The application of our tool prototypes to the above specifications has lead to the results that are shown in Table 5 for the four different levels of available information (cf. Sect. 2).

Table 5. Results of Feature Interaction Detection for Complex Building Control Systems

Available Information	Number of Feature Interactions			Number of Points of Interaction		
	Floor32	**Floor32X**	**Δ**	**Floor32**	**Floor32X**	**Δ**
1. Requirements	34	41	7	52	58	6
2. Strategies (Signal Types only)	34	41	7	52	58	6
3. Object Structure	32	38	6	47	53	6
4. Environment	38	44	6	63	69	6

Typical interactions that have been identified in *Floor32* at the requirements level were between $\{U2, U3, U4, FM1, FM6\}$ as well as between $\{UH2, FMH2\}$. These interactions are fairly obvious as the interacting needs describe different aspects of a common feature, which is a consequence of the fact that needs were specified very fine-grained and solution-oriented. This also explains the relatively huge number of interactions that can be identified.

When the system was extended, seven new feature interactions have been introduced on the requirements level. One of these interactions occurs between $\{UA2, UA3, FA1\}$, which is inside the alarm system domain. Additionally, interactions between needs of different domains have been identified; e.g., $\{U2, UA10\}$. This interaction occurs, because both need $U2$ and need $UA10$ employ means of lighting for their realization.

The number of interactions that have been identified at level 1 and level 2 is the same. This can be attributed to the fact that our tool prototypes currently do not consider additional interactions caused by a coupling on the strategy level. Further, no interactions from level 1 can be eliminated because no point of interaction exists that just *produces* signals. One reason for that is that all sensors in our system can receive special signals that change a sensor's mode from polling to event-based communication.

From level 2 to level 3, the number of possible interactions is reduced by three and the number of the particular points of interactions is lessened by five. One example for an interaction that is eliminated is the one between $\{U2, U3, U4, UA2\}$. This interaction is detected at levels 1 and 2 because the lighting control object type $RoomLt$ as well as the control object type $RoomOcc$, which is used for the alarm system, each aggregate an instance of $Contact$ (cf. Fig. 8).

Finally, by using information about the system's environment at level 4, six additional interactions have been uncovered. As an example, one of these interactions occurs between $\{FM1, U2, UH6\}$, because an interaction between the control object type $TempSens$, which is needed for realizing the temperature control need ($UH6$), and the control object type $BlindAct$, which realizes the lighting control needs ($FM1$ and $U2$), is present.

3.3 Tool Complexities

The tool prototype that has been employed for the above calculations is part of our *PROTAGOnIST* tool set [25]. It has efficiently been developed using the programming language Java. Each type of entity is represented by a Java class and each type of relation is implemented by attributes as well as the required accessor and mutator methods. Thus, operations on product model data, like following the *realizedBy* relations for identifying the point of interaction, can easily be implemented by directly mapping abstract concepts from the model level to Java code.

For performing the above case studies, tools with a total number of 10700 lines of code have been developed. Approximately 1900 lines of this code are needed for the actual interaction detection. An additional 4100 lines of code are required for parsing the development documents [26]. The remaining code, which has automatically been generated from UML class diagrams, realizes the class frames as well as the attributes and relations of the product model.

An evaluation of the run-time complexity of this tool, supported by statistical measurements, shows very moderate processing power requirements.

For example, to determine feature interactions at the requirements level (see Sect. 2.1), for each of the n needs, all tasks that realize this need have to be examined. Based on exemplary evidence, this number does not seem to correlate with the number of needs. This implies that a linear run-time complexity of $O(n)$ can be attained for this activity.

For the concrete example of *Floor32X*, the actual detection of the interactions that are depicted in Table 5 has consumed 2.5 s of processing time on a 440 MHz HP-PA RISC workstation.

4. Conclusion and Perspectives

The extension and reuse of existing systems are important activities in software development. To correctly carry out these activities, the developers need to be aware of the interactions that exist between features. This paper has shown an approach for the detection of feature interactions that is based on a formal model of the development products. With this approach, detailed information about feature interactions can automatically be derived from existing requirements specification documents at each stage of the proposed requirements engineering method. The more complete this development information is, the more refined the information about the interactions will be.

Our approach can also be used for guiding the developer in such a way that undesirable interactions between features can be avoided. In detail, this means that after each important step in the requirements specification process (e.g. after important tasks have been elicited), possible interactions can be computed, and the developers can decide on how to handle undesirable interactions.

At the time of writing, our approach has been realized for four important activities in our requirements specification method: the specification of needs and tasks, the creation of the object structure, the specification of the communication between strategies, which realize tasks, and the modelling of the environment.

A further activity in this method is the precise specification of the behavior of the strategies through extended finite state machines, which basically are finite state machines that have been extended by variables and control constructs. With an extension of the product model to reflect these types of state machines, a far more refined examination of feature interactions might become feasible. Currently, we are in the process of refining our product model for that purpose, which will be followed by an evaluation of a possible refinement of our feature interaction detection approach.

Acknowledgements

Parts of this work have been funded by the Deutsche Forschungsgemeinschaft DFG in the Special Collaborative Research Center SFB 501 “Development of Large Systems with Generic Methods”.

Finally, we want to thank the anonymous referees, who—by providing detailed and constructive comments—have contributed to the improvement of this paper.

References

[1] V. Hartkopf, V. Loftness. "Global Relevance of Total Building Performance" in *Automation in Construction*. 8. New York; Amsterdam: Elsevier Science. 1999

[2] A. Metzger, S. Queins. "Specifying Building Automation Systems with PROBAnD, a Method Based on Prototyping, Reuse, and Object-orientation" in P. Hofmann, A. Schürr (Eds.) *OMER – Object-Oriented Modeling of Embedded Real-Time Systems*. GI-Edition, Lecture Notes in Informatics (LNI), P-5. Bonn: Köllen Verlag. 2002. pp. 135–140

[3] B. Ramesh, M. Jarke. "Towards Reference Models for Requirements Traceability" in *IEEE Transactions on Software Engineering*, 27(1). 2001. pp. 58–93

[4] E. Magill, M. Calder (Eds.) *Feature Interactions in Telecommunications and Software Systems VI*. Amsterdam: IOS Press. 2000

[5] D. Amyot, L. Charfi, N. Gorse et al. "Feature Description and Feature Interaction Analysis with Use Case Maps and LOTOS" in *Sixth International Workshop on Feature Interactions in Telecommunications and Software Systems FIW '00*. Glasgow, Scotland. 2000

[6] R.J.A. Buhr. "Use Case Maps as Architectural Entities for Complex Systems" in *IEEE Transactions on Software Engineering, Special Issue on Scenario Management*, 24(12). 1998. pp. 1131–1155

[7] T. Bolognesi, E. Brinksma. "Introduction to the ISO Specification Language LOTOS" in *Computer Networks and ISDN Systems*, 14. 1986. pp. 25–59

[8] L. du Bousquet. "Feature Interaction Detection using Testing and Model-checking – Experience Report" in *World Congress in Formal Methods*. Toulouse, France. 1999

[9] M. Calder, A. Miller. "Using SPIN for Feature Interaction Analysis – A Case Study" in M.B. Dwyer (Ed.) *Model Checking Software 8th International SPIN Workshop*. Toronto, Canada. 2001. pp. 143–162

[10] M. Heisel, J. Souquieres. "Detecting Feature Interactions – A Heuristic Approach" in G. Saake, C. Turker (Eds.) *Proceedings 1st FIREworks Workshop*, Preprint 10/98. Fakultät für Informatik, University of Magdeburg. 1998. pp. 30–48

[11] R. van der Straeten, J. Brichau. "Features and Feature Interactions in Software Engineering using Logic" in *Feature Interactions in Composed Systems*. 2001. pp. 79–88

[12] C. Araces, W. Bouma, M. deRijke. "Description Logics and Feature Interaction" in *Proceedings of the International Workshop on Description Logics DL-99*. 1999

[13] E. Ernst. "What's in a Name?" in *Feature Interactions in Composed Systems*. European Conference on Object-Oriented Programming ECOOP 2001. Workshop #8. 2001. pp. 27–33

[14] E. Pulvermüller, A. Speck, J.O. Coplien et al. "Feature Interaction in Composed Systems" in (Eds.) E. Pulvermüller, A. Speck, J.O. Coplien et al. (Eds.) *Feature Interactions in Composed Systems*. European Conference on Object-Oriented Programming ECOOP 2001. Workshop #8. 200. pp. 1–6

[15] A. Olsen, O. Færgemand, B. Møller-Pedersen et al. *System Engineering Using SDL-92*. 4th Edition, Amsterdam: North Holland. 1997

[16] A. Metzger, S. Queins. "Early Prototyping of Reactive Systems Through the Generation of SDL Specifications from Semi-formal Development Documents" in *Proceedings of the 3rd SAM (SDL And MSC) Workshop*. Aberystwyth, Wales: SDL Forum Society; University of Wales. June, 2002

[17] S. Queins. *PROBAnD – Eine Requirements-Engineering-Methode zur systematischen, domänenspezifischen Entwicklung reaktiver Systeme*. Ph.D. Thesis. Department of Computer Science. University of Kaiserslautern. 2002

[18] R. Budde, K. Kautz, K. Kuhlenkamp, et al. *Prototyping: An Approach to Evolutionary System Development*. Berlin; Heidelberg: Springer-Verlag. 1992

[19] A. Metzger. *Small Building Control Example*. On-line Development Documents. Department of Computer Science. University of Kaiserslautern. 2003
http://wwwagz.informatik.uni-kl.de/d1-projects/ResearchProjects/InteractionExample/

[20] ITU-T. *User Requirements Notation (URN) — Language Requirements and Framework*. Recommendation Z.150. Geneva: Iternational Telecommunication Union, Study Group 17. 2003
http://www.usecasemaps.org/urn/

[21] A. Mahdavi, A. Metzger, G. Zimmermann. "Towards a Virtual Laboratory for Building Performance and Control" in R. Trappl (Ed.) *Cybernetics and Systems 2002*. Vol. 1. Vienna: Austrian Society for Cybernetic Studies. 2002. pp. 281–286

[22] G. Zimmermann. “Efficient Creation of Building Performance Simulators Using Automatic Code Generation” in *Energy and Buildings*, 34. Elsevier Science B.V. 2002. pp. 973–983

[23] S. Queins, G. Zimmermann: *A First Iteration of a Reuse-Driven, Domain-Specific System Requirements Analysis Process*. SFB 501 Report 13/99. University of Kaiserslautern. 1999

[24] S. Queins, M. Trapp, T. Brack et al. *Floor 32*. On-line Development Documents. Department of Computer Science. University of Kaiserslautern. 2002
http://wwwagz.informatik.uni-kl.de/d1-projects/ResearchProjects/Floor32/

[25] A. Metzger et al. *PROTAGOnIST – Tools for Automated Software Development*. Web Site. Department of Computer Science. University of Kaiserslautern. 2003
http://wwwagz.informatik.uni-kl.de/staff/metzger/protagonist/

[26] A. Metzger. “Requirements Engineering by Generator-Based Prototyping” in H. Alt, M. Becker (Eds.) *Software Reuse – Requirements, Technologies and Applications*. Proceedings of the International Colloquium of the SFB 501. Department of Computer Science. University of Kaiserslautern. 2003. pp. 25–35