ELSEVIER

Available online at www.sciencedirect.com
SCIENCE DIRECT®

Computer Networks 45 (2004) 625–644

COMPUTER NETWORKS

www.elsevier.com/locate/comnet

Feature interactions in embedded control systems

Andreas Metzger *

Department of Computer Science, University of Kaiserslautern, P.O. Box 3049, 67653 Kaiserslautern, Germany

Available online 23 March 2004

Abstract

Present-day software systems, like modern telecommunications systems or distributed Internet applications, have reached levels of complexity that can no longer be addressed with ad-hoc techniques. Therefore, systematic approaches for coping with this huge complexity are required. To correctly develop such systems, interactions between the system's features have to be considered, as these might be the origin of incorrect system behavior. In addition to the traditional domains of large systems, increasingly complex systems can be found in the domain of embedded control systems, of which automotive as well as building and home control systems are interesting examples. In particular, the environment, in which an embedded control system operates, plays a crucial role in such a system's behavior, and can thus be the source for additional interrelationships between features. In this article, a systematic approach for the automatic detection of feature interactions in embedded control systems is presented, which allows the identification of interactions within a system as well as the detection of interactions that are caused by the environment.
© 2004 Elsevier B.V. All rights reserved.

Keywords: Embedded system; Requirements engineering; Feature interaction

1. Introduction

Embedded control systems have already become ubiquitous in today's world, where these systems are used in a multitude of application domains. These domains range from controllers for household appliances (each washer or microwave is equipped with some form of computer control), to automotive control systems (where systems like the anti-lock braking system, ABS, or the electronic stability program, ESP, are standard equipment in each new car), to systems that are employed for controlling large facilities (like office buildings, hotels, and airports). With an ever-increasing demand for new functionality and the number of devices that should be controlled, the complexity of these systems rises. Modern luxury cars, like BMW's 7 series, are equipped with up to 70 controllers and house software of more than 60 MB [1]. State-of-the-art building automation systems take a large number of different physical effects (light, temperature, humidity, etc.) into account to attain optimal performance [2], and like the control system of the Burj Al Arab hotel in Dubai, control up to 26,000 data-points [3].

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

* Tel.: +49-631-205-3142; fax: +49-631-205-2162.
E-mail address: metzger@informatik.uni-kl.de (A. Metzger).

1389-1286/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.comnet.2004.03.002

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. Another problem can be discovered when parts of such complex systems are to be reused, because *required* interrelations to other parts of the reused system have to be taken into account.

In the telecommunications domain, this *feature interaction problem* [4] has been recognized for a long time and has received considerable attention from the scientific community as can be seen by events like the "Feature Interaction Workshop", which has been held for the seventh time in 2003 [5]. However, this feature interaction problem has so far not been specifically approached for embedded control systems—probably because only now has the complexity reached dimensions that requires systematic control and resolution.

As a notable difference with respect to the telecommunications domain, embedded control systems will almost always be embedded in a physical environment, which provides responses to a system's stimuli and which is not part of the actual software system. Because of this special role of the environment, the identification and resolution of interactions that occur *within* the system is not enough, but also interrelationships that are introduced by interactions with the environment must be considered.

An example for such special interactions can be found when a building automation system for controlling lighting and heating is considered. At first glance, the heating control part can be realized independently of the lighting control part. However, if sunlight is employed for establishing the desired level of illumination, the controlled space might considerably heat up, which will present an interaction with the temperature control part, as it has no means for avoiding this situation and thus might not be able keep the desired temperature. A further example for this important role of the environment comes from the automotive domain. Let us assume that a car is equipped with cruise control and the electronic stability program (ESP) [6]. If cruise control speeds up the car at a high rate on a wet road, the wheels will slip. This is detected by the ESP, which intervenes by braking the slipping wheels, thus avoiding undesirable vehicle dynamics (like skidding). If the cruise control component knew nothing of the ESP feature and the physical reasons for its interventions, it would continue accelerating the car, which inevitably would result in the car leaving the road.

Before being able to treat feature interactions, they must first be detected. In this paper, we suggest detection concepts that work on requirements specification documents, which provide an early detection of interactions and reduce the effort for resolving such interactions in later development phases. Because of the complexity of the systems under consideration, such a detection activity cannot reasonably be performed manually. Therefore, an automation of this activity is required. This article elaborates on concepts and solutions for such automation that have been first presented in [7]. Most notably this includes the extension of the approach from building automation systems to the broader domain of embedded control systems and the refinement of the detection algorithms based on experience with first prototypes of the detection tools.

In the remainder of this article, the product model that classifies the elements of the requirements specification documents is introduced in Section 2. Based on the entities of this product model, our concepts and algorithms for detecting feature interactions in embedded control systems are presented in Section 3, followed by the results of four case studies in Section 4. Finally, the approach is discussed and related to existing work in the feature interaction field (Section 5).

2. The product model

For automating any development activity, explicit knowledge of the development process (in the form of explicit models) must be available. A *product model* is one such type of model that explicitly describes the development artifacts (or products) and the relations between them for a given development method (see [8]). We will therefore employ such a product model for the feature interaction detection concepts that are presented in this article.

To provide a better understanding of the entities of this product model, we introduce this section with a sketch of our requirements specification method that produces the types of artifacts that are specified in the product model.

2.1. The requirements specification method

In Fig. 1, an overview of the documents and activities of our requirements specification method *PROBAnD* [9,10] is provided.

System development with the PROBAnD method starts from a *problem description*, which is divided into an *environment description* and a collection of *needs*.

The environment description contains a description of the environment's structure, which—in the case of building control systems—could include a floor-plan showing that the building is made up of one floor with three rooms. Further, sensors and actuators, which resemble the interface of the control system to its environment, are depicted. This is usually done in natural language; e.g., in a text that states: "There is one illumination sensor, one motion-detector as well as one light in each of the rooms".

From this environment description, an initial *control system structure* can be attained, which is usually refined in subsequent steps. To handle the huge number of control objects that needs to be considered for large systems, *control object types* are formed, which are aggregated according to the hierarchy of the environment's objects. In the building automation domain and in related domains, this has proved to be a suitable approach. The reason for this is that *control objects* (the objects of the control system) can most often be identified with objects in the environment. For example, the control object that is responsible for controlling an engine of a car can be derived from the engine object in the environment. A possible refinement would be the introduction of an injection control object, which contributes to the overall engine control object, and which aggregates the ignition actuators (spark plugs) and an oxygen sensor (required for emission reduction).

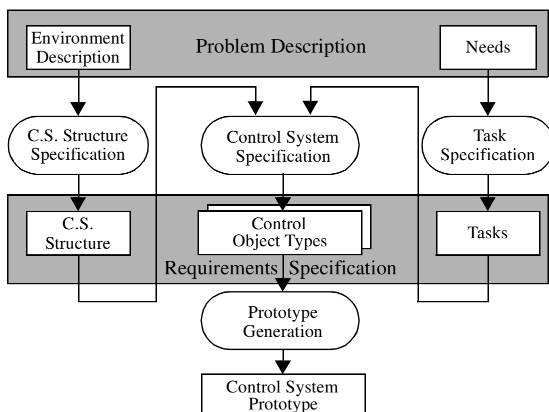

Fig. 1. Documents and activities of the PROBAnD method. Arrows depict input/output dependencies.

As introduced above, the other part of the problem description is made up of a collection of needs. These needs describe the requirements from the point of view of the users, expressed in natural language. From these needs, *tasks*, which resemble developer requirements, are derived in such a way that each task can be assigned to a single control object type. This allows for the unambiguous traceability of responsibilities (i.e., tasks) to control object types.

After the control object types have been identified and tasks have been assigned, *strategies* for realizing the tasks are provided. These strategies specify the behavior of a task in an operational style using (partial) finite state machines. The behavior of the complete control object type is achieved by composing the respective partial state machines. For testing purposes, prototypes can automatically be created from such a set of control object types, by generating intermediate specifications [11] in SDL (Specification and Description Language [12]).

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 operational 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 [13]. During the design phase, different solutions for the strategies are possible and legitimate. This implies

that when moving from the analysis to the design phase, a change of the feature interactions can occur. Therefore, a detection (and if necessary treatment) of these changed interactions should be performed to ensure system quality.

2.2. A small building control example

To illustrate the application of the PROBAnD method for specifying embedded control systems and to provide an example for the feature interaction detection concepts that will follow, a small building control system is presented in the following paragraphs.

Let us assume that the users have the four needs that are listed in Table 1. The first need (N1) expresses the users' wish to have an automatic control of the illumination inside a room. The second need (N2) reflects energy-optimization criteria. The third need (N3) is requested to guarantee undisturbed working conditions. And finally, need N4 represents a temperature control requirement.

According to the above activities of the requirements specification method, these needs are refined, leading to a collection of tasks, which are assigned to specific control object types. The aggregation structure of possible control object types, which have been established using the respective building description, is shown in Fig. 2.

Table 1
Needs of 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

The top level controller node *RoomCtrl* corresponds to the building object *room*. As a refinement of this structure, the remaining controllers (*IllumCtrl*, *GlareCtrl*, and *TempCtrl*) have been identified with the physical effects they should control, which are *illumination*, *glare* and *temperature*. Sensors and actuators can typically be found as leaf nodes in such a structure; e.g., *IllumSens* or *BlindAct*. In Fig. 2 two instances of *IllumSens* can be found. This reflects the fact that, for illumination control, the current illumination inside the room is required, where for glare control, we need to determine the amount of (sun-)light that reaches the workplace.

Table 2 lists suitable tasks for refining the above needs. One such refinement for need N3 is provided by tasks T6 and T3, which reflect the developer's decision of avoiding glare by employing the blinds.

2.3. The product model of the PROBAnD method

After having introduced the PROBAnD method and having provided an example of its application, the method's product model is presented in this subsection. In order to employ the product model for an automated approach, this model has to be machine readable, meaning that there must be a way of accessing—and if necessary modifying—the instances of the product model in a systematic and algorithmic way. The use of object-oriented metamodels can provide the required level of formality for reaching this goal [14,15]. Entities in this metamodel describe *types* of development products (like the environment specification documents or the set of control object type documents).

To further classify these types and to define the permissible relations between them, a further

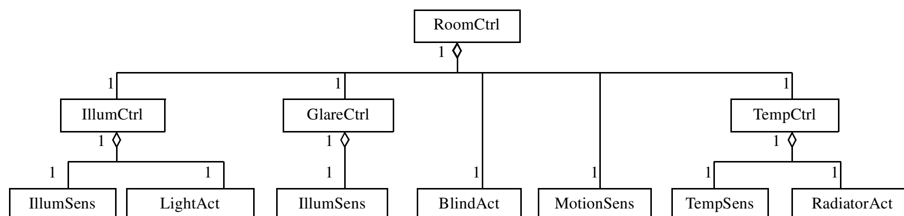

Fig. 2. Object structure of a small building control system for heating and lighting.

Table 2
Task list of a small building control system

Task	Description	Realizes	implemented By
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

level of abstraction (i.e., a meta-metamodel) is defined, whose entities are thus *metatypes* of development products. The most general metatype of a development product is classified as an *artifact type*. On the level of the requirements specification, each artifact has a unique name and can carry a more detailed description. More special metatypes of development products are *atomic artifact types*, which classify development products that cannot be decomposed any further. Examples for atomic artifacts are attributes or needs. Atomic artifacts contain the actual development information independent of its representation and as such are based on an abstract syntax (cf. [16]). Consequently, different *view types* can exist for each atomic artifact, which contain the information of an atomic artifact in a concrete representation (based on a concrete syntax). Additionally, the metatype *configuration type* classifies artifacts that aggregate less complex ones. Finally, *document types* in this classification are special configurations, which aggregate views only.

This classification of types of development products provides a means for simplifying interaction detection. As atomic artifacts represent the same development information as their views do, the detection concepts can work by employing the more abstract atomic artifacts and thus the concepts benefit from the abstraction provided by these artifacts. As a prerequisite, the set of available development documents is parsed to instantiate the atomic artifacts and their relations. This step is carried out by decomposing the concrete documents into views, followed by extracting the actual atomic artifact information from these views. Such a decomposition is possible because the development documents expose a precise structure (e.g., in the form of tables like Table 2). This step is similar to a programming language compiler constructing an abstract syntax tree from source code. Details on how this parsing of requirements specification documents is performed can be found in [11,17].

The atomic artifacts of the PROBAnD method (which are instances of the atomic artifact type of the meta-metamodel) are shown in Fig. 3. These entities present the set of abstract development information that is used for our feature interaction detection process.

Needs are user requirements and as thus are modelled as a specialization of *requirement*. Only needs that represent functional requirements, i.e., *functional needs*, are considered during our requirements specification process. Because of their granularity and meaning, we will identify these functional needs with features for the remainder of this article.

Needs are realized by *tasks*, which are developer requirements and as thus are also modeled as a specialization of the atomic artifact *requirement*. Each task can be realized by other tasks, which is reflected by the *realizedBy* relation between the atomic artifacts *requirement* and *task*. This relation—like all the other relations of the product model—are explicitly reflected in the development

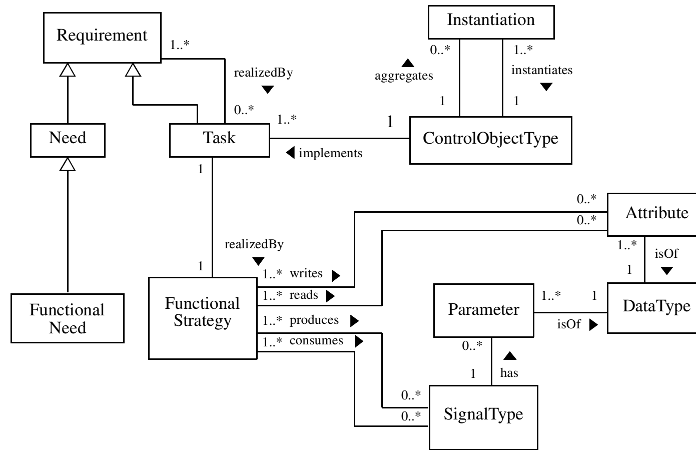

Fig. 3. Atomic artifacts of the product model of the PROBAnD method.

documents, which are used for instantiating the atomic artifacts.

Each task describes a responsibility that has to be fulfilled by the one *control object type* that implements this very task. This unambiguous assignment of responsibilities to tasks is described by the one-to-many *implements* relation from the atomic artifact *control object type* to the atomic artifact *task*. As it has been depicted in Section 2.1, 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 atomic artifact *control object type* and the atomic artifact *instantiation*.

For each task, *a functional strategy* is specified. As has been pointed out, such a strategy describes a possible solution for realizing the responsibility (or the behavior) 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 because of modelling guidelines of the PROBAnD method, 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.

3. Feature interaction detection

In Section 2.3 we have already identified functional needs with features. Thereupon, the goal of detecting feature interactions in embedded control systems can be reached by identifying dependencies between functional needs. These dependencies can be extracted by following the relations between atomic artifacts in an instantiation of the product model. Depending on the available 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. The order of these levels resembles the usual order of the development activities in our PROBAnD method.

3.1. Detection at requirements level

Early in the requirements specification process, needs and tasks are the only atomic artifacts that will have been specified. Therefore, interactions at this level can only be identified by employing these

atomic artifacts together with the *realizedBy* links between them. From this data, a dependency graph between requirements can be attained. In such a graph, *points of interaction* can be identified, which are nodes that realize more than one need and have more than one direct parent. From these points of interaction, the actual interactions can be deduced.

Fig. 4 shows the dependency graph for the small building control example (Section 2.2). According to Table 2, the needs N1 and N2 are realized by task T1, where T1 is realized by the tasks 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. With the knowledge of such points of interaction, we are able to determine a feature interaction between the needs N1 and N2 (at T1) as well as interactions between N1, N2 and N3 (at T3, T4 and T5).

To algorithmically determine the points of interaction, an empty set S_i is created for each task T_i $(i = 1, \ldots, n)$. Then, for each need N_j $(j = 1, \ldots, m)$ we follow all *realizedBy* relations from this need to all tasks that realize this need and add N_j to the set S_k for each task T_k that is traversed. At the end, a task T_i $(i = 1, \ldots, n)$ can be identified as a point of interaction if $|S_i| > 1$ and T_i is realizing more than one requirement (i.e., there are more than one *realizedBy* links ending at T_i).

Since the first implementation of the above concept, the experience that we have gathered in applying the detection tools has lead us to the conclusion that interactions that are caused by needs that are *directly* realized by the task at the point of interaction do not present a critical situation. An example for such an interaction is the one between needs N1 and N2. Such interactions are caused by a too fine-grained specification of needs or by orthogonal needs (like need N2, which states an energy saving requirement that needs to be added to the basic lighting control requirement). Therefore, a subsequent step in the detection process is the removal of such kinds of interactions, which can be done easily by checking if all realized requirements at a given point of interaction are needs.

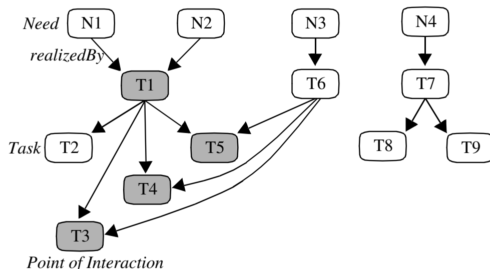

Fig. 4. Dependency graph of the requirements of the small building control system.

It should further be noted that all potential interactions are identified at this level. However, some of these interactions will not necessarily occur in the final system. One example for such a condition is a task at a given point of interaction that will never be used for realizing more than one need simultaneously during run-time.

3.2. Detection at object structure level

To refine the above set of potential 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, which will be available as soon as the object structure has been specified, provides clues as to whether an interaction can occur based on the instantiation of control object types. As this instantiation information refines the knowledge about the usage of a control object type (and its implemented tasks), some interactions might be eliminated from the list of possible interactions. Such elimination is correct if the dependent tasks are not "used" at the same point of instantiation.

An example for such a situation is illustrated in Fig. 5. As can be derived from the specification documents of the small building control system, task T5 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 *IllumSens* and therefore will employ only this instance for realizing the required functionality, there will be no interaction between tasks T1 and T6 at point T5.

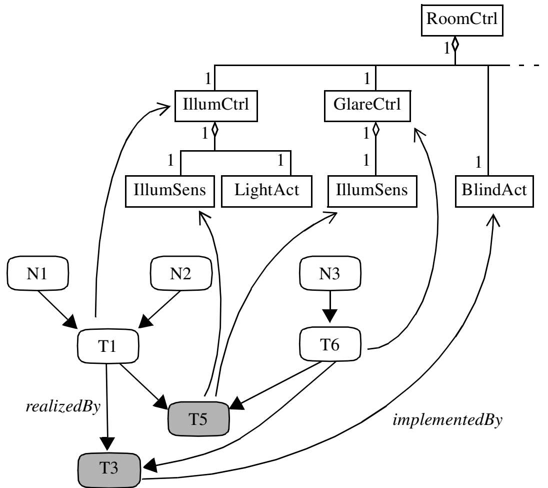

Fig. 5. Elimination of impossible interactions by using object structure information.

To identify such situations, we make use of a modeling guideline of the PROBAnD method, which requires that control object types that implement tasks that are connected by *realizedBy* relations should be instantiated as closely as possible. The reason for this is a reduction of the number of signals that have 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 have to be considered.

Let $D(T_i)$ be the set of all tasks that are directly realized by T_i, and $P(T_i, T_j)$ the set of all control object types that are on the shortest path between the object type that implements task T_i and the one that implements T_j (the algorithmic determination of these paths can be found in [7]). If we compute the sets $P(T_i, T_j)$ for all tasks T_j that are in $D(T_i)$, then there can be no interaction between the tasks T_j at T_i if there are no common object types along the paths except the control object type at the point of interaction. This is the case if no intersection between any of two sets $P(T_i, T_j)$ leads to a set holding more than the control object type that implements T_i. When computing these intersections, a set $P(T_i, T_j)$ that only contains one control object type is ignored, as this implies that the interacting tasks are implemented by the very same object type.

For the above example of task T5, $D(\mathrm{T5})$ is $\{\mathrm{T1}, \mathrm{T6}\}$, and thus $P(\mathrm{T5}, \mathrm{T1}) = \{IllumSens, IllumCtrl\}$ and $P(\mathrm{T5}, \mathrm{T6}) = \{IllumSens, GlareCtrl\}$, which leads to the intersection $\{IllumSens\}$. Hence, there is no interaction at point T5.

Whereas in the case of T3 (see Fig. 5) the following sets are computed: $P(\mathrm{T3}, \mathrm{T1}) = \{BlindAct, RoomCtrl, IllumCtrl\}$ and $P(\mathrm{T3}, \mathrm{T6}) = \{BlindAct, RoomCtrl, GlareCtrl\}$, which leads to the intersection $\{BlindAct, RoomCtrl\}$ and therefore this interaction cannot be eliminated from the list of potential interactions.

The same is valid for the point of interaction at T4 (not shown in Fig. 5).

3.3. Detection at strategy level

After the above levels of information have been considered, dependencies between tasks that are introduced by their realization can be examined as soon as the developers have specified the strategies of the respective tasks. Dependencies on this level can occur because strategies can be coupled by signals or attributes to exchange information (see Section 2.3).

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 that 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 atomic artifact *functional strategy* to the atomic artifact *signal type* (resp. *attribute*). However, experience has shown that this might obscure real interactions if the development documents are incomplete; e.g., the information that an attribute is also written by a second task might not have been specified. Therefore, we refrain from implementing the above heuristics in our current version of the interaction detection tool.

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 a developer to specify that the strategies of requirements that do not participate in a *realizedBy* relationship operate on the same set of data. This is realized by having these strategies read and write identical attributes, thus introducing a coupling between the respective tasks. If in our small control system we introduced a need N5 that was realized by a task T10 for setting the desired value for the illumination level, the strategy of this task would be implemented in such a way that a new desired level is written to the attribute that is used for storing this value. Task T1 in turn would be reading this attribute for determining if the required level of illumination has already been reached, thus an interaction between N1 and N5 would be noted.

In fact, such a situation points to inconsistent requirements specification documents, as the dependency that is introduced by this coupling reflects a *realizedBy* relationship between tasks that has not been specified explicitly. In the above example, T1 relies on the setting of the desired illumination level, whereby T1 is realized by T10.

As a consequence of this observation, our detection process makes the coupling through strategies explicit by creating a hypothetical *realizedBy* link between the tasks under consideration, which then allows the identification of the new interactions by using the algorithm introduced in Section 3.1. However, the introduction of such hypothetical links is feasible only for attributes because their scope is limited to that of a single control object type. If one tries to extend this approach to signals (which appear just to be a different means of communication between tasks), the problem arises that one cannot easily limit the scope of signals that are instances of the same type and therefore cannot clearly determine the *realizedBy* relationships. Assuming there was a very general signal type named *acknowledge* that can be sent by any control object type to notify of the reception of a signal, an interaction caused by all tasks that employ this signal type would be falsely identified.

3.4. Detection at environment level

As mentioned in Section 1, the physical environment of an embedded control systems plays a crucial role in its behavior, because this environment can be the source of an implicit coupling between different parts of the system. This fact can already be discovered in our small building control example. The dependency graph in Fig. 4 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 this is that sunlight can be a considerable source of heat. Consequently, an interaction between N4 and the other needs will be noted in the deployed system.

Therefore, it is important to consider such kinds of interactions in the process of detecting feature interactions. To automatically discover these interactions, the physical interrelationships in the environment must be made explicit, which requires knowledge about a system's environment. To attain models that reflect this knowledge, we rely on the fact that during the development of each reactive system, a simulator of the system's environment will eventually be needed for testing the dynamic behavior of the system before deployment. As such a simulator has to consider the physical interrelationships for correctly simulating the environment, the simulator's models will contain the dependencies that are needed for the detection of feature interactions. In our case, such simulators have been modelled using the PROBAnD method. Examples are building performance simulators [18] or a vehicle dynamics simulator [19].

The interface of a control system to its environment is realized by its sensors and actuators. A sensor measures physical values of the system's environment; an actuator is responsible for interfering with the environment, which most often results in changes of physical values. Ergo, for each sensor or actuator of the control system, a matching counterpart must exist in the simulator that provides the respective values or the expected behavior. To easily find such matching counterparts when coupling a control system and a

simulator, we create object types with identical names in both systems [20]; e.g., the *TempSens* control object type of the small building control system will have a *TempSens* counterpart in the simulation model that is responsible for providing the simulated temperature inside the room. Therefore, each task that is implemented by a sensor or actuator control object type in the control system will eventually (although indirectly) be realized by one or more tasks of the respective simulator object type. As the dependencies between the simulator tasks reflect the physical interrelationships, detecting the points of interaction using these tasks will lead to interactions caused by the environment.

This solution is sketched in Fig. 6 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 Ta, that the task for simulating the temperature sensor is Tb, and if we further assume that both Ta and Tb are realized by task Tc, which is responsible for simulating the room temperature, then this very task Tc can be identified as a point of interaction. This leads to the conclusion that N1, N2 and N3 expose an interaction with N4 through the physical environment.

To automatically detect these interactions, the instantiation of the product model of the control system and that of the simulation are systematically merged to achieve a combined model instance. During this merging process, sensor and actuator control object types are employed as connection points between these two systems (their names are identical as it has been pointed out above). For each pair of *implementedBy* links from controller tasks to control object types and from object types to simulator tasks, a hypothetical *realizedBy* link is created analogous to Section 3.3. Fig. 7 shows the result of such a merge for the example above.

After this has been performed, the initial algorithm from Section 3.1 can again be employed without any modification and will return the interactions and the corresponding points of interaction caused by the environment.

Usually, the developers should be aware at which point in *their* system (i.e., control system) the interaction occurs. Therefore, a final step in interaction detection is tracing back from the point of interaction in the environment to the tasks that implement the control object types at the connection points. In the small example, this leads to the identification of the interactions of $\{N1, N2, N3, N4\}$ at T3 and T9.

Interestingly, the detection of additional interactions through the environment again might point to an incomplete specification (like in the case of Section 3.3), because without explicitly reflecting the interaction in the control system there can be no way that this physical interaction

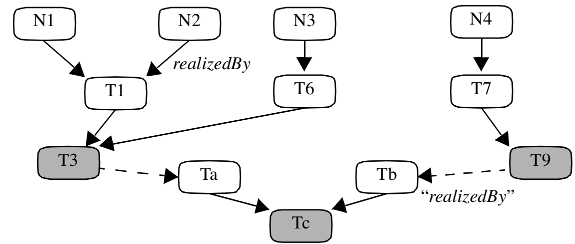

Fig. 7. Merged product model instances and the detected interactions on this level.

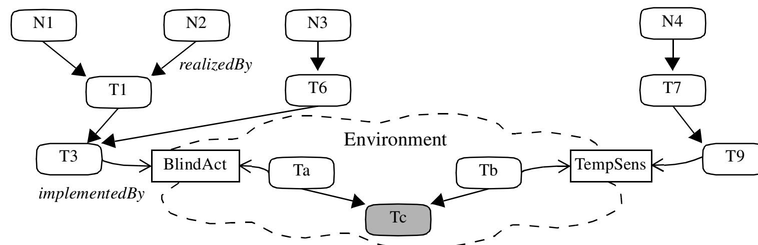

Fig. 6. Dependency between lighting and heating components that is introduced by the physical environment.

will be handled correctly; e.g., in our small example there should be a task that knows of this interaction and consequently only allows the usage of daylight if this is not in conflict with the temperature control requirement (i.e., this task would implement some form of conflict resolution).

4. Case studies

To evaluate the above concepts and to examine their feasibility in a real development context, tool prototypes have been implemented and applied in several case studies that are introduced in the following subsections. In these case studies, systems for different embedded control domains have been evaluated for their feature interactions and the numbers of the detected interactions are given below. These systems include two building, one automotive, and one railway crossing control system.

As noted above, each level in our detection approach refines the detected feature interactions. Therefore, the numbers of interactions for a given level neither represent interactions that are mutually exclusive from the previous level (interactions at the requirements level can be present at the environment level) nor do these numbers necessarily include the interactions of the previous level (an interaction that was detected at the requirements level can be eliminated at the object structure level).

It should further be noted that although the detection of interactions is possible at all development stages, the results of the case studies have been determined after the systems had been completely specified (i.e., the specifications already existed before our tools were developed). Therefore, we think it quite natural to assume that the resulting systems (and numbers of interactions) would have been different if our tools had been used during development.

4.1. Building automation systems

In the first case study, a heating and lighting control system, which we call *Floor32* in the remainder of this article, was used as an example. A detailed description of this system and an analysis of qualitative and quantitative development data can be found in [21]. To give an idea, a few of the system's 67 needs for heating and illumination are provided in Table 3.

In *Floor32* a total of 233 tasks and 37 control object types were specified for realizing the needs. 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 [22]. This is reflected by 12 additional needs, some of which are listed in Table 3. Further, 19 additional tasks and three control object types have been added for realizing the new functionality.

The number of interactions that have been identified is shown in Table 4. No interaction detection at the level of strategies has been performed for *Floor32*, because no attributes had been specified in the requirements documents. Therefore, in order to be able to compare the two case studies, the same kind of results is shown for *Floor32X* in addition to the complete detection results.

Typical interactions that have been identified in *Floor32* at the requirements level were $\{U2, FM1, FM6\}$ as well as $\{UH2, FMH2\}$. 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 fine-grained and solution-oriented. This also explains the relatively large number of interactions that can be identified.

When the system was extended (*Floor32X*), seven new feature interactions were introduced on the requirements level. One of these interactions occurs between UA2 and 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 needs U2 and UA10 employ means of lighting for their realization. This indicates the problem that, as soon as an alarm is triggered, the chosen light scene—as requested in U2—can no longer be maintained, which is in conflict with the original requirement.

From level 1 to level 2, the number of possible interactions is reduced by two and the number of

Table 3
Needs of large building control systems (excerpt)

Domain	Need	Description
Illumination		
Floor32 + *Floor32X*	U2	As long as the room is occupied the chosen light scene has to be maintained
	FM1	Use daylight to achieve the desired illumination whenever possible
	FM6	The facility manager can turn on/off any light in a room or hallway section
Heating		
Floor32 + *Floor32X*	UH2	The comfort temperature shall be reached as fast as possible during heating up and shall be maintained as best as possible afterwards
	FMH2	The use of solar radiation for heating should be preferred against using the central heating unit
Alarm		
Floor32X	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_{alarm} seconds. Otherwise, the alarm must be triggered
	UA10	If an alarm is triggered, all lights in the corresponding sections are turned on. When the alarm is reset, the lights 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

Table 4
Results of feature interaction detection for large building control systems

Level of information	Number of feature interactions			Number of points of interaction		
	Floor32	*Floor32X*		*Floor32*	*Floor32X*	
Requirements	18	21		25	27	
Object structure	17	19		22	24	
Strategies	–	–	41	–	–	74
Environment	23	26	44	52	54	91

the particular points of interactions is lessened by three. One example of an interaction that is eliminated is the one between U2 and UA2. This interaction is detected at the requirements level because the control object type that is responsible for lighting as well as the control object type that is realizing parts of the alarm system, each aggregate a (different) instance of the control object type *Contact*. For lighting control, *Contact* is used as a sensor to determine if the lamps have successfully been turned on. For the alarm system, a contact sensor is used for checking if a window has been opened.

Finally, by using information about the system's environment at level 4, three additional interactions have been uncovered in *Floor32X*, As an example, one of these interactions occurs between FM1 and UH2, because an interaction between the control object type *TempSens*, which is needed for realizing the temperature control need UH2, and the control object type *BlindAct*, which realizes the lighting control needs FM1, is present.

It should be pointed out that the numbers in Table 4 differ from the numbers that were originally published in [7], because the described improvements of the detection process (like the elimination of interactions caused by the direct realization of needs) have been incorporated.

4.2. Automotive control system

A far less complex case study comes from the automotive domain. The system that has been evaluated implements a comfort function that provides the smooth stopping of a car even if the brake pedal is not softly released at the end of the braking process. Thereupon, the needs of this *SmoothBrake* system can be stated briefly (see Table 5). N1 represents the actual feature of the system. Need N2 provides a required level of traffic safety, and N3 allows manual system override. In addition to the original specification, need N4 has been added, which requires the manual setting of the speed threshold for distinguishing between regular and smooth brake operation.

For realizing the system, 16 tasks were specified and assigned to seven control object types. A further discussion of this case study and an evaluation of its dynamic behavior under different situations can be found in [23]. The result of the feature interaction detection is shown in Table 6.

The first two interactions that are identified are $\{N1, N2\}$ as well as $\{N3, N2\}$, as expected, since need N2 is the traffic safety need that should always be guaranteed. At the strategy level, a new interaction between N4 and N2 is introduced. This can be attributed to the fact that the strategies of tasks that realize N4 write the speed threshold attribute that is read by the strategies that realize N2 to determine the driving situation for ensuring safe operation. Finally, two more interactions are introduced by the environment. These are caused by the physical coupling between the brake actuator and the speed sensor.

Table 5
Needs of automotive control system

Need	Description
N1	Ensure that car stops smoothly when braking
N2	Traffic safety must not be adversely impacted
N3	Manual system override should be possible
N4	It should be possible to manually set the speed threshold

Table 6
Results of feature interaction detection for automotive control system

Level of information	Number of feature interactions	Number of points of interaction
Requirements	2	4
Object structure	2	4
Strategies	3	5
Environment	5	9

4.3. Railway crossing controller

The final case study originated as a design contest for the SDL Forum's SDL and MSC Workshop in 2002 [24]. The goal was to develop a railway crossing controller (including the required parts of the environment) for controlling the gate according to different control strategies; e.g., one strategy stated that cars should take precedence if there are too many cars waiting at the gate.

Further, a dynamic number of tracks and trains had to be realized. This last requirement led to an extension of our PROBAnD method to implement dynamic instantiations. However, for feature interaction detection, we simplified that requirement by modelling the fixed number of five tracks for the crossing controller.

Our initial solution [25] contained 13 needs, 61 tasks as well as 21 control object types. For detecting feature interactions, we have isolated the control system part (realized by 17 tasks and six control object types) and have applied our tool. The results are shown in Table 7.

Other than the four interactions on the requirements level that again can be attributed to fine-grained needs, this system shows no additional interactions that are caused by the environment. This is because the environment has been considered in detail during the specification of the

Table 7
Results of feature interaction detection for railway crossing controller

Level of information	Number of feature interactions	Number of points of interaction
Requirements	4	10
Object structure	4	10
Strategies	4	12
Environment	4	12

control part to achieve optimal control strategies, which for example requires that the interrelationship between the state of the gate (opened/closed) and the cars (which have to wait when the gate is closed) is considered.

5. Discussion and related work

Some of the interactions that have been identified above do not seem to be critical (for example {U2, FM1, FM6} in *Floor32*, Section 4.1) and therefore should probably not be considered in the detection process, thus refining the set of interactions. Such an elimination of interactions might be suitable if an existing system is extended, and therefore only the introduction of *undesirable* (and thus critical) interactions with old parts of the system has to be determined (for example {U2, UA10} in *Floor32X*). However, for other development activities this might not be the case.

Especially when reusing parts of an existing system for constructing a new system, a detailed knowledge of *required* interrelationships (which are also feature interactions) is absolutely necessary. Otherwise, important parts of the system needed for the "reused" behavior might be removed, which would lead to unexpected results. To give an example for such a *required* interaction, let us assume that—as a consequence of detecting the interaction {N1, N2, N3, N4} in Section 3.4—we have extended our small control system in such a way that the blinds are closed when sunlight is heating up the room, thus introducing an interaction between N1, N2, N3, and N4 at the requirements level. If this system now should be reused as a temperature control system only, all parts that are relevant for lighting (lights, blinds, etc.) can obviously be removed. However, if this removal is exercised without considering the required interaction between the temperature controller (N4) and the blind (N2 and N3), a malfunctioning system is the result, as the heating up of the room due to daylight cannot be stopped.

Unfortunately, distinguishing between required and undesirable interactions is not trivial and thus cannot be automated without further information. Therefore, we refer this distinction to the developer. Still, the reduction of complexity for feature interaction detection is enormous. In *Floor32X*, there were "only" 21 interactions at 27 points of interaction that had to be examined at the requirements level, compared to 331 requirements (with 298 *realizedBy* links) that had to be inspected manually for interactions without the support of our automated approach. To give an impression of the potential complexity of such a detection activity, Fig. 8 shows a dependency graph for 103 requirements of *Floor32X* (the points of interaction are highlighted).

As already pointed out in Sections 3.3 and 3.4, a careful investigation of the specification documents is indicated if the number of feature interactions between the different levels of information increases, as this might point to deficiencies in the underlying documents, like missing *realizedBy* links between tasks or missing conflict resolution tasks and strategies.

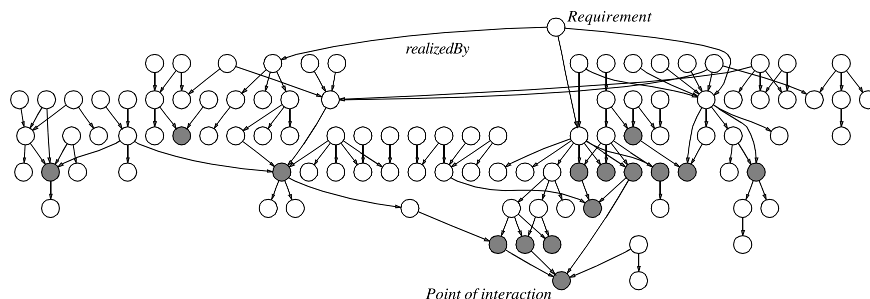

Fig. 8. Dependency graph of 103 of the 331 requirements of the control system of case study *Floor32X*.

5.1. General applicability

Different levels of information have been presented above, as an input to the interaction detection approach. These different levels also provide a means for extending the applicability of our approach to more general domains and requirements specification techniques. Table 8 gives an overview of the dependency on the domain and the specification technique for the approach at different levels.

At the requirements level, a prerequisite for our detection algorithm to be applicable is the existence of a traceability relation between needs (user requirements) and tasks (developer requirements). An example of a specification language that fulfils this requirement is the Goal Oriented Requirement Language (GRL [26]), which provides the notion *of goal* (equalling need) and *task* as well as a *means-end* relation between goals and tasks, which equals the *realizedBy* relation.

The method for detecting interactions with knowledge of the object structure strongly relies on the properties of the application domain. Thus, the current algorithm cannot directly be applied to systems that require dynamic instantiation or object structures that deviate from a tree-like structure. However, as only a traceability relation from tasks to the control object types is needed from the specification technique, other domains that expose similar object structures can be suitable candidates for the above approaches.

To determine the contribution of signals and attributes to the existing set of interactions, a traceability relation from the strategies (or tasks) to the used signal types and attributes is absolutely necessary. This is a very strong restriction, and therefore implies a strong dependency on the specification technique. Additionally, only asynchronous (or event-based) systems, which communicate using signals, are considered. No synchronous (method-call-based) systems have so far been studied.

Finally, at the environment level, the approach will only be suitable for embedded systems (that interact with an environment). Plus, without the existence of environment simulators (or environment models) that have been specified with the same specification technique as that of the control system, the presented detection concept cannot work.

The most important prerequisite for all of the above levels to be applicable is a traceability relation from requirements (needs) to the artifacts that realize them. These relationships all contribute to the "post requirements specification traceability", which is identified by Gotel and Finkelstein in [27]. As the authors correctly observe, most development methods provide some form of support for that, be it in a user-configured way in general-purpose tools or by an explicit support in special workbenches. Although some of the more specialized tools provide automated support for creating traceability links, most of the information has to be provided by the developers (e.g., which requirement is currently realized by the ongoing activity). Therefore, the developers should be encouraged to create and maintain explicit traceability links. Unfortunately, these links do not seem to directly support the final product and therefore there may be a lack of motivation of the developers for doing this, we believe however that as soon as the benefits of having such information are well understood, this motivation will increase. We have seen that these benefits include automatic feature interaction detection and the

Table 8
Applicability of detection approach depending on level of information

Level of information	Dependency on domain	Dependency on specification technique	General applicability
Requirements	O	+	+
Object structure	++	+	O
Strategies	+	++	O
Environment	++	++	–

automatic and effortless identification of parts of the system that are affected by changes in user requirements.

As it is always with manual tasks, these are prone to error. In our case, forgetting to create a *realizedBy* link might hide actual interactions; creating wrong links might introduce false ones. The latter error is not problematic, as when looking into the interaction in more detail, a wrong relationship between requirements can be discovered. The first kind of error is more problematic, as it might go unnoticed. However, as we have pointed out in Section 3.3, the detection algorithm at the strategy level can unveil such missing links.

As our detection process can be applied no matter how much information has been specified, an interaction detection can be performed at any time in the development process. This allows for quality control measures to be executed whenever suitable. We think that an ideal process would identify interactions already at the beginning, when only needs and tasks have been specified, to identify possible conflicting requirements and to resolve such conflicts. When system development proceeds, more refined detection results can be expected as well as a refinement of the already detected ones at the requirements level.

Such a repetitive application of the detection algorithms can also be used for identifying required and undesirable interactions. When an extension activity has been performed, one should carefully compare the interactions that have been found before and after that step to identify undesirable ones. For a reuse activity, one should compare the interactions in the resulting system with the ones in the reused system and check whether necessary interactions have been removed.

5.2. Tool development

For realizing the detection tools, we have chosen a technique that exploits the model characteristics of the metamodel (“a metamodel is a *model* of a model”). Thus, standard modelling techniques and tools can be employed. We use iLogix’s Rhapsody [28] for generating Java classes from product model entities in such a way that each class provides a way of accessing the attributes and relations of the artifact it represents. With these Java classes as a basis, we can automate the detection concepts by mapping the abstract algorithms on the model level to Java methods.

To achieve overall efficiency in feature interaction detection, the effort for manual detection of feature interactions needs to be compared with the effort for the creation of the automatic tools. This is similar to the reuse problem, i.e., reuse only pays off if the initial effort for creating the reusable asset is less than the total effort for creating the asset from scratch for each new project.

Unfortunately, we have not yet been able to perform explicit measurements of the effort (i.e., time) required for manual feature interaction detection. Therefore, we can only provide an estimate of the efficiency of our automated approach by providing information on the size of the tools that have been developed. Assuming that there is a good correlation between size and effort, we could deduce the effort for tool creation and compare this with the effort of manual detection.

In Table 9, the number of non-comment source statements (NCSS) for the interaction detection tool is shown (NCSS is a more accurate metric than simply counting the lines of code (LOC) [29]). Besides the total number of NCSS, the individual numbers for the implementation of interaction detection at each of the different levels of information are listed. The size of the class frames and the code for accessing attributes and relations is left out since these were automatically generated by Rhapsody. We believe these numbers—and consequently the effort for tool creation—to be very moderate.

Table 9
Size of the different parts of feature interaction detection tool for various levels of information

Level of information	Size of implementation (non-comment source statements)
Requirements	110
Object structure	175
Strategies	110
Environment	120
Sum	515

Further, our tool has very modest processing time requirements. For the concrete example of *Floor32X*, the detection of the interactions used less than one minute of CPU time on a 440 MHz HP-PA RISC workstation (including I/O).

The tool prototype that has been employed for interaction detection is part of our *PROTAGOnIST tool* set [30], which also includes the tool for parsing the development documents (approximately 2,000 NCSS [17]) to generate the abstract artifacts of the product model on which the detection tool works.

5.3. Related work

Wilson and Magill [31] are among the few authors who consider feature interactions in embedded control systems by investigating home automation systems. They recognize the importance of the environment for such systems, and therefore focus on the devices (sensors and actuators) within such an environment. A run-time (or on-line) solution for the interaction problem is suggested that employs device managers—which are in fact feature managers [4]—that discard any requests that will lead to feature interactions. To realize such device managers, a model of the environment is taken into account, in which the physical variables and the kind of influence (positive or negative) of each device are specified. By employing concepts from the operating system domain (like mutual exclusion), different conflict resolution strategies can be implemented within these device managers. This on-line approach provides a great flexibility when new devices are added to the system. However, if these new devices influence physical variables that have so far not been considered or if they influence existing variables in an unknown fashion, the device managers have to be redesigned (which is an off-line activity that requires the extension of the environment models).

For telecommunications systems, several feature interaction detection approaches have been presented in the literature. Especially the proceedings of the "Feature Interaction Workshops" [5,32] or the ECOOP workshop [33] and the overview papers by Calder et al. [4] and Keck and Kuehn [34] provide good references to interesting contributions in this area.

These contributions can be classified into ones that use some form of system specification as input and others that employ the actual implementation code for feature interaction detection. Contributions of the latter kind are for example presented by Bousquet [35], Ernst [36] and Blair and Pang [37]. However, these approaches can only be applied late in the development process, and therefore undesirable interactions might have already found their way into the final product, where removing or resolving these interactions can become very costly.

Therefore, in our opinion, solutions that employ models (or specifications) should be preferred. A comprehensive list of these solutions can be found in [4]. A possible solution is the one suggested by Amyot et al. [38], where undesirable interactions are identified by employing Use Case Maps [39] and LOTOS [40]. After features have been semi-formally described with Use Case Maps, these features are formally defined with LOTOS, which can then be used as input for verification.

Many of the suggested model-based approaches require the modeller to explicitly and formally specify features. In contrast to this, our method does not force the requirements engineer to provide any *additional* information during the specification process. The existing documents can be used exactly as they are in our requirements specification method *without* considering feature interaction. Therefore, this approach can provide an increase in product quality with only little additional effort.

Because of the nature of telecommunications systems, the environment plays a lesser role in interaction detection in these systems. Therefore, the above approaches will hardly be applicable for our problem of detecting feature interactions caused by physical couplings, unless an extension of these approaches is considered.

6. Conclusion and perspectives

Important activities in software development are the extension and reuse of existing systems. To

correctly carry out these activities, the developers need to be aware of the interactions that exist between features. This article has shown an approach for the detection of feature interactions that is based on a metamodel of the development products. By specifying the detection concepts at the model level (such as evaluating requirements graphs), detailed information about feature interactions can automatically be derived from existing requirements specification documents at each stage of the proposed requirements specification method. In particular, once environment simulators are available, important information about interactions caused by the environment of the system can be obtained.

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

Such an effortless computation of feature interactions could easily be used for computing design metrics for evaluating the quality of the models. An example of such a metric is the number of interactions in relation to the number of requirements. Depending on the domain that is considered and the development method that is employed, a high ratio might point to a bad system design as too many fine-grained requirements might have been specified. A good starting point for establishing and validating such metrics can be the case studies that have been presented here.

So far, our method has been implemented offline. This was possible as the static aggregation hierarchy of the control object types and the static communication dependencies between strategies prevented unexpected changes in the system. However, when moving away from such static structures (e.g., in order to reflect dynamic usage and reconfiguration scenarios in large office buildings or smart homes), an investigation of online (i.e., run-time) techniques [41] may be beneficial.

Acknowledgements

I am very grateful to Christian Webel, who has contributed considerably to the implementation of the interaction detection tool and has participated in writing the initial paper [7].

I further wish to thank the anonymous referees and also the participants and reviewers of the 7th "Feature Interaction Workshop", who have contributed to the improvement of this work by providing detailed and constructive comments.

Parts of this work have been funded by the Deutsche Forschungsgemeinschaft DFG under grant SFB 501 "Development of Large Systems with Generic Methods".

References

[1] A. Saad, Prototyping bei der BMW Car IT GmbH, JavaSPEKTRUM 2 (2003) 49–53.

[2] V. Hartkopf, V. Loftness, Global relevance of total building performance, Automation in Construction 8 (4) (1999) 377–393.

[3] R. Boehme, D. Covell, S. Grant, et al., Automated control systems, Green Hotelier Magazine 22 (2001).

[4] M. Calder, M. Kolberg, E. Magill, et al., Feature interaction: a critical review and considered forecast, Computer Networks 41 (2003) 115–141.

[5] D. Amyot, L. Logrippo (Eds.), Feature Interactions in Telecommunications and Software Systems VII, IOS Press, Amsterdam, 2003.

[6] H. Bauer, K.-H. Dietsche, J. Crepin, et al. (Eds.), BOSCH Automotive Handbook, fifth ed., Robert Bosch GmbH, Stuttgart, 2000.

[7] A. Metzger, C. Webel, Feature interaction detection in building control systems by means of a formal product model, in: D. Amyot, L. Logrippo (Eds.), Feature Interactions in Telecommunications and Software Systems VII, IOS Press, Amsterdam, 2003, pp. 105–121.

[8] B. Schätz, A. Pretschner, F. Huber Franz, et al., Model-based development, Technical Report TUM-I0204, Department of Informatics, TU Munich, Munich, 2002.

[9] 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, Lecture Notes in Informatics (LNI), Köllen Verlag, Bonn, 2002, pp. 135–140.

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

[11] A. Metzger, S. Queins, Model-based generation of SDL specifications for the early prototyping of reactive systems, in: M.E. Sherrat (Ed.), Telecommunications and Beyond: The Broader Applicability of SDL and MSC, Lecture Notes in Computer Science, Vol. 2599, Springer, Heidelberg, 2003, pp. 158–169.

[12] A. Olsen, O. Færgemand, B. Møller-Pedersen, et al., System Engineering Using SDL-92, fourth ed., North Holland, Amsterdam, 1997.

[13] R. Budde, K. Kautz, K. Kuhlenkamp, et al., Prototyping: An Approach to Evolutionary System Development, Springer, Heidelberg, 1992.

[14] C. Atkinson, Meta-modeling for distributed object environments, in: Z. Milosevic (Ed.), Proceedings of 1st International Enterprise Distributed Object Computing Conference (EDOC'97), IEEE Computer Society, 1997, pp. 90–103.

[15] T. Clark, A. Evans, Foundations of the unified modeling language, in: D. Duke, A. Evans (Eds.), Proceedings of Bcs-Facs Northern Formal Methods Workshop, Ilkley, UK, September 1996, Springer, Heidelberg, 1997.

[16] D. Harel, B. Rumpe, Modeling languages: syntax, semantics and all that stuff—Part I: The basic stuff, Report MCSOO-16, Weitzman Institute, Rehovot, Israel, 2000.

[17] A. Metzger, Requirements engineering by generator-based prototyping, in: H. Alt, M. Becker (Eds.), Proceedings of the International Colloquium of the SFB 501, Software Reuse—Requirements, Technologies and Applications, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, 2003, pp. 25–35.

[18] G. Zimmermann, Efficient creation of building performance simulators using automatic code generation, Energy and Buildings 34 (2002) 973–983.

[19] G. Calderón Meza, Modeling friction in a ground vehicle—brake discs & tire/road friction models, in: A. Metzger, G. Zimmermann (Eds.), Modellierung reaktiver Systeme: Ein Fallbeispiel, SFB 501 Report 08/03, University of Kaiserslautern, Kaiserslautern, 2003, pp. 37–50.

[20] A. Mahdavi, A. Metzger, G. Zimmermann, Towards a virtual laboratory for building performance and control, in: R. Trappl (Ed.), Proceedings 16th European Meeting on Cybernetics and Systems Research (EMCSR), Cybernetics and Systems 2002, vol. 1, University of Vienna, Vienna, April 2002, pp. 281–286.

[21] 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, Kaiserslautern, 1999.

[22] S. Queins, M. Trapp, T. Brack, et al., Floor 32, on-line development documents, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, 2004. Available from <http://wwwagz.informatik.uni-kl.de/d1-projects/ResearchProjects/Floor32/>.

[23] J. Brandt, B.H. Schäfer, SmoothBrake—A prototyping case study, in: A. Metzger, G. Zimmermann (Eds.), Modellierung reaktiver Systeme: Ein Fallbeispiel, SFB 501 Report 08/03, University of Kaiserslautern, Kaiserslautern, 2003, pp. 51–60.

[24] A. Williams, R. Probert, Q. Li, et al., The winning entry of the SAM 2002 design contest: a case study of the effectiveness of SDL and MSC, in: R. Reed (Ed.), SDL 2003: System Design, Lecture Notes in Computer Science, Vol. 2708, Springer, Heidelberg, 2003, pp. 387–403.

[25] S. Queins, A. Metzger, The PROBAnD railway crossing specification, in: SDL-2000 Design Contest of the 3rd SAM (SDL and MSC) Workshop, Aberystwyth, Wales, June 2002. Available from <http://wwwagz.informatik.uni-kl.de/staff/metzger/publications.html#QuM02>.

[26] D. Amyot, Introduction to the user requirements notation: learning by example, Computer Networks 42 (3) (2003) 285–301.

[27] O.C.Z. Gotel, A. Finkelstein, An analysis of the requirements traceability problem, in: Proceedings of 1st International Conference on Requirements Engineering, IEEE Computer Society Press, Los Alamitos, CA, 1994, pp. 94–101.

[28] B.P. Douglass, Real-Time UML—Second Edition: Developing Efficient Objects for Embedded Systems, Addison-Wesley, Reading, MA, 2000.

[29] R.B. Grady, Practical Software Metrics for Project Management and Process Improvement, Prentice Hall, Englewood Cliffs, NJ, 1992.

[30] A. Metzger et al., PROTAGOnIST—Tools for Automated Software Development, web site, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, 2004. Available from <http://wwwagz.informatik.uni-kl.de/staff/metzger/protagonist/>.

[31] M. Wilson, E. Magill, An environmental model for service interaction in home networks, in: Proceedings of Postgraduate Research Conference in Electronics, Photonics, Communications and Software (Prep) 2003, Exeter, UK, April 2003.

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

[33] 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, 2001.

[34] D.O. Keck, P.J. Kuehn, The feature and service interaction problem in telecommunications systems: a survey, IEEE Transactions on Software Engineering 24 (10) (1998) 779–796.

[35] L. du Bousquet, Feature interaction detection using testing and model-checking—experience report, in: World Congress in Formal Methods, Toulouse, France, 1999.

[36] E. Ernst, What's in a name? in: 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, 2001, pp. 27–33.

[37] L. Blair, J. Pang, Aspect-oriented solutions to feature interaction concerns using AspectJ, in: D. Amyot, L. Logrippo (Eds.), Feature Interactions in Telecommunications

and Software Systems VII, IOS Press, Amsterdam, 2003, pp. 87–104.
[38] D. Amyot, L. Charfi, N. Gorse, et al., Feature description and feature interaction analysis with Use Case Maps and LOTOS, in: E. Magill, M. Calder (Eds.), Feature Interactions in Telecommunications and Software Systems VI, IOS Press, Amsterdam, 2000.
[39] R.J.A. Buhr, Use Case Maps as architectural entities for complex systems, in: Special Issue on Scenario Management, IEEE Transactions on Software Engineering 24 (12) (1998) 1131–1155.
[40] T. Bolognesi, E. Brinksma, Introduction to the ISO specification language LOTOS, Computer Networks and ISDN Systems 14 (1986) 25–59.
[41] M. Calder, M. Kolberg, E. Magill, et al., Hybrid solutions to the feature interaction problem, in: D. Amyot, L. Logrippo (Eds.), Feature Interactions in Telecommunications and Software Systems VII, IOS Press, Amsterdam, 2003, pp. 295–312.

Andreas Metzger studied Computer Science at the University of Kaiserslautern in Germany, where in 1998 he received his diploma degree. Since then he has been working as a research and teaching assistant at the Department of Computer Science of the University of Kaiserslautern, where most of his research has been conducted in the context of the special collaborative research center SFB 501 "development of large systems with generic methods" of the German science foundation DFG. In 1999 he has been a visiting scientist at the Department of Architecture at Carnegie Mellon for two months. His research interests include metamodeling and automated software engineering and he is interested in persuing a career in academia. Andreas is currently finalising his Ph.D., in which he is investigating in how far quality improvements in software development can be achieved by automating activities of the early development phases (requirements engineering).