Modeling Structure and Behavior for Knowledge-Based Software Configuration

Christian Kühn[*]

Abstract. There are several approaches to knowledge-based configuration, which are successfully applied in technical domains. Nevertheless they appear not to be sufficient for configuring software-based systems, as they primarily proceed in a structure-based manner, but do not take the system's behavior into account. In this context, the configuration of software-based systems does not mean the programming of software, but the composition of existing software modules into an individual software variant. Our approach uses state-based behavior descriptions for domain objects, which can be directly used for making decisions during the configuration process. Our goal is to apply this approach for a knowledge-based configuration of future software-based vehicle electronic systems, which implement customer-individual vehicle functions.

1 INTRODUCTION

The configuration of complex software-based systems which can carry out a high number of variants can be a difficult and time-consuming task. In this context, configuration does not mean the programming of software, but the composition of existing components into an individual software variant. In the area of Artificial Intelligence, various methods for configuration have been developed that could already be successfully applied, especially for technical systems. The majority of these methods are structure-based, i.e. the configuration process as well as the specification of the configuration objectives are based on the structure of the system to be configured. For the configuration of software-based systems these structure-based approaches seem to be suitable, but not sufficient. In particular, the consideration of the software system's behavior appears to be very significant. What software *does* can be more important than how it is structured. There are existing approaches, which in principle take behavior during configuration into account. However the application of behavioral knowledge is rather limited to the evaluation of solutions or partial solutions. An extension of modeling techniques is needed, to adequately enable model software behavior, which can be utilized for configuration.

Once again it must be reiterated that our approach is not aimed at the process of individual software development or programming. It is intended for building up a high number of individual software systems which are based on the same set of basic components or modules. At the time of configuration the process of module development has already been finished. Furthermore our method is intended rather to configure combined hardware/software systems (embedded systems) than pure software systems. Nevertheless we will focus on the software aspect in the following. Software can be considered as a domain, while software modules are understood as domain objects and will be referred to as concepts in the knowledge base (software module concepts).

[*] DaimlerChrysler AG, Research and Technology, HPC T721, D-70546 Stuttgart, Germany, e-mail: christian.kuehn@daimlerchrysler.com

In the following section (sec. 2) a short introduction into current knowledge-based configuration approaches will be given with a focus on structure-based and behavior-based techniques. The limits of these approaches will be pointed out. Section 3 describes our proposed extension of current structure-based configuration knowledge modeling by behavior models, which serve as a basis for the configuration process (sec. 4). In section 5 the configuration of software-based vehicle electronic systems will be illustrated as an example of application. This paper ends with a summary and brief outlook (sec. 6).

2 APPROACHES TO KNOWLEDGE-BASED CONFIGURATION AND THEIR LIMITS FOR SOFTWARE CONFIGURATION

Knowledge-based configuration belongs to the class of synthesis tasks. For both the representation of knowledge and the reasoning process different methodologies have been developed. The following examples are the most relevant (see [10]):

- *Structure-based approach:*
  In the structure-based approach a compositional, hierarchical structure of the domain objects serves as a guideline for the control of problem solution.
- *Constraint-based approach:*
  Representing restrictions between objects or their properties, resp. by constraints and evaluating these by constraint propagation. The constraint-based approach is not in competition with the structure-based approach but is frequently combined with it.
- *Resource-based approach:*
  Resource-based configuration is based on the following principle, that interfaces between components are specified by the exchanged resources. Components make a number of

resources available and also consume resources themselves. A task specification exists in form of required resources.

- *Case-based configuration:*
  Case-based problem-solving methods are therefore identified, so that knowledge concerning already-solved tasks is saved and is used for the solution of new tasks. The reason for this is that similar tasks lead to similar solutions.

Next to these approaches for knowledge-based configuration there are several other techniques, like rule-based techniques and especially techniques that combine different methods, e.g. using simulation, optimization or spatial reasoning throughout the configuration process. Some of these approaches which look at behavior, will be dealt with in the following subsection (2.2). We will now consider the structure-based approach for software configuration.

2.1 Structure-based configuration

Under the term configuration, the step by step assembly and parameter setting of objects from an application domain to a problem solution or configuration is understood, where certain restrictions and given task objectives should be fulfilled (see [10], [17]).

A configuration problem comprises the following components (see [10]):

- A set of objects in the application domain and their properties (parameters).
- A set of relations between the domain objects, while taxonomic and compositional relations are of particular importance for configuration.
- A task specification (configuration objectives) which specifies the demands a created configuration must fulfill.
- Control knowledge about the configuration process.

The process of configuration consists of a sequence of configuration steps. As the domain model describes the set of all possible solutions (configurations), each configuration step can limit this set until it is reduced to a final solution. In order to distinguish between the knowledge representation level and the solution level, the objects on the knowledge representation level will be called domain objects or concepts, and the solution elements will be called instances of these concepts. Possible configuration steps are specialization (refining an instance to a more specific concept), decomposition (top-down instantiation of an aggregate's components), integration (bottom-up including an instance into an aggregate), and parameterization (determining an instance's property value). Examples of configuration systems based on these types of configuration steps are PLAKON [4], KONWERK [6], [8], and EngCon [1].

An overview of the technologies, applications and systems is given in the above mentioned literature.

2.2 Utilizing behavior models for configuration

Some new approaches take the dynamic system's behavior over time into consideration; this concerns principally the integration of simulation techniques in knowledge-based configuration (see e.g. [3], [9], [15], [16]). The aim on one hand is to reduce partial solutions and calculate values through simulation. On the other hand simulation can be used for testing the behavior of a fully or partly configured system and therefore facilitates the verification with the subsequent evaluation of (partial) configurations.

For verification and evaluation of a configuration or partial configurations the results of other (heuristic) problem solving methods are verified through simulation. This can be integrated within a configuration system in the following ways (see [9]):

- Mapping of the knowledge base and the partial configuration onto a suitable model for simulation,
- Simulation of this model and
- The evaluation of the simulation through the configuration system.

There are simulation procedures which are integrated into configuration both in the area of quantitative continuous simulation (e.g. [9], [15], [16]) and in the area of condition-based, qualitative simulation (e.g. [3]).

2.3 Limits of the current techniques for software configuration

The above described structure-based techniques seem to be suitably applicable to configure complex software-based systems, as these systems are based on a module structure. Although these techniques provide a good basis for configuring software systems, they are not yet sufficient as most of them are confined to the structural construction of systems without taking the systems' behavior into consideration. The reactive system behavior can become very complex, especially for software-based systems.

An area of application similar to configuration, is the planning that also belongs to the class of synthesis tasks. For planning, time and temporal aspects play a central role (see [2]). However in planning, the target object is a sequence of operations, which convert a starting state into a target state. In contrast to configuration, here the organization of the different planning steps along a temporal axis is of great importance. However in configuration the aim is to compose a system (i.e. finding a structure), which in our context has a desired reactive behavior. Both tasks are similar, but the focus is different.

Often in the context of the "software" domain, a clear module structure which can be transferred to the knowledge base has already been worked out during the software development process. In contrast to this, specific knowledge about properties and behavior[1] of the software modules have to be supplemented.

[1] In this context behavioral knowledge does not mean the detailed behavior e.g. on code level, but abstract knowledge about the behavior that can be utilized for configuration.

Although on principle the above-mentioned approaches to the integration of simulation into the knowledge-based configuration take the behavior of a configuration system into account, they are not sufficient for the development of complex, software-based systems, since they are not designed for the high variance of potential system behavior. Instead of this, a modeling is needed which allows an adequate, abstract software modeling, which makes direct conclusions possible from the modeled behavior of software objects (meaning software modules as domain objects).

Therefore we propose an approach which forms the basis of a combination of structural configuration knowledge and behavior-based configuration knowledge. (The structural knowledge means domain objects with properties, which are organized in taxonomic and compositional hierarchies. The behavior-based knowledge being statecharts allocated to the domain objects.)

3 EXTENDING CONCEPT HIERARCHIES BY ABSTRACT SOFTWARE BEHAVIOR MODELS

The aim is to model knowledge of the usable software modules for configuration, so that reasoning from this knowledge is possible. This is described in the demands on modeling techniques and problem solving methods for software configuration in [14]. A declarative, generic modeling of software modules as abstract concepts is proposed. These concepts (as well as the other components), are arranged in a concept hierarchy and are provided with attributes. For example, several concepts of the same software module can be specified, but with different ranges of detail. Likewise, the same concept can be described through its generic representation of several software modules, when these modules are represented by an abstract superconcept because of common properties.

Of course, with a description of a software module by a domain object (a so-called concept), there are numerous properties that can be specified, e.g. development data (version, authors, short documentation), hardware allocation (suitable processors, required memory of the module (ROM), required memory during runtime (RAM), applied periphery components, properties for application (input/output interfaces). We want to extend these “simple” concept properties for software concepts by behavior models as complex concept properties.

There are different ways of describing behavior. A simple way to describe the behavior of a software module, is by using the explicit allocation of one or more buzzwords which describe the eligibility and possible applications of the module, e.g. *light activation* or *light dimming*.

The functionality of a module can be represented in more depth by state models, e.g. finite state machines or Petri nets, just as with rules. There are different possibilities. It is not intended to model the behavior as detailed as on code level, but to give an abstract description of the behavior. In the following, statecharts [13] are used as a description method for the behavior of modules, which we will go into in more depth in the subsequent sections.

3.1 Assigning state behavior to domain objects

Statecharts are an extensive method, their functionality is described in detail in [13]. In contrast to state machines, statecharts allow (among others):

- Modularization (that means that single states can be refined to statecharts themselves),
- Parallelism (that means that several states can be active at the same time, i.e. states can be orthogonal) and
- Real-time conditions.

Especially in the context of embedded systems, statecharts are often used for system (and software) specification. In this way statecharts are frequently used for the generation of executable code (e.g. C or C++) for embedded systems (see [5], [11], [13]). In contrast to this, in our approach statecharts serve exclusively for the abstract description of (software) behavior as a starting point for selecting existing software modules and appropriately combining them. The aim is not to generate code (see above).

The same state-based behavior, which is described in our approach by statecharts, can equally be described with other techniques, e.g. with finite state machines, regular expressions, or the explicit set of all traces or sequences of state changes over time (as can be seen in figure 2). While the modeling by the user or knowledge engineer is generally simpler and more adequate on statechart level[2], the internal handling of behavior is however simpler using one of the other techniques – e.g. finite state machines – as operations such as comparison or specialization are easier to perform. Thus each modeled statechart can be converted into an internal representation before it is used in the configuration process.

In the approach described here, only parts of the general statechart concepts are utilized. It concerns state transition diagrams, whose transitions are labeled with triggering conditions. Corresponding to this a state can have transitions to several target states, whereas transitions are flagged with different conditions. Conditions can base on external events or on internal events (for example entry into or exit from states). In addition such events can be combined by Boolean operators with further conditions, e.g. with information about the activity or inactivity of other states. The usable operators for describing conditions (in addition to the Boolean operators *and, or, not*) are listed in table 1.

[2] In particular, statecharts are fundamentally concerned with finite specifications, whilst traces can be infinite. In comparison to finite state machines, statecharts are much more compact. Regular expressions are too difficult to handle for the user.

operator	description
in(instance, state)	The specified state has to be active.
some-in(set of instances, state)	The specified state has to be active for one of the instances.
all-in(set of instances, state)	The specified state has to be active for all of the instances.
entry(instance, state)	Transition to the specified state takes place.
some-entry(set of instances, state)	Transition to the specified state takes place for one of the instances.
all-entry(set of instances, state)	One instance passes into the specified state, afterwards all instances of the set are in this state.
exit(instance, state)	Transition from the specified state takes place.
some-exit(set of instances, state)	Transition from the specified state takes place for one of the instances.
all-exit(set of instances, state)	One instance exits the specified state, afterwards no instance of the set is in this state.
timeout([event], duration)	Transition is triggered by a real-time condition: after the specified event has happened (or after entering the current state, if no event is given), and the given duration of time has past.
extern	Transition is triggered external.

Table 1. Operators for specifying conditions for state transitions

Statecharts are assigned to the individual domain objects (in this case software modules) as properties of these components. For this, state variables with a set of permissible states (which are alternatively, i.e. exclusively, active) can be assigned to the domain objects. In our approach the statecharts of different concepts can be based on the same states, i.e. a domain object's statechart can also use other domain objects' states, or at least reference other domain objects' states in its conditions for transitions. To do this a domain pattern can be specified which describes the referenced states.

An important point about configuration is that during the configuration process it is not necessarily definite, which domain objects are relevant for the solution, i.e. which will be instantiated as a part of the solution (consider the distinction of concepts and instances in chapter 2). To access any set of instances' states in the statecharts we introduce predicate logical condition operators for transitions (some-in, all-in, some-entry, all-entry, some-exit, all-exit). Figure 1 shows a state transition taking place, if at least one vehicle door is opened or all doors are closed, as an example.

A classification in a concept hierarchy requires that objects can be specialized, decomposed, and parameterized together with their behavior (see above). Whereas parameterizing the behavior models does not play a significant role in our approach, we are focusing on specializing modeled behavior (in the taxonomical hierarchy) and decomposing modeled behavior (in compositional hierarchies) in the following.

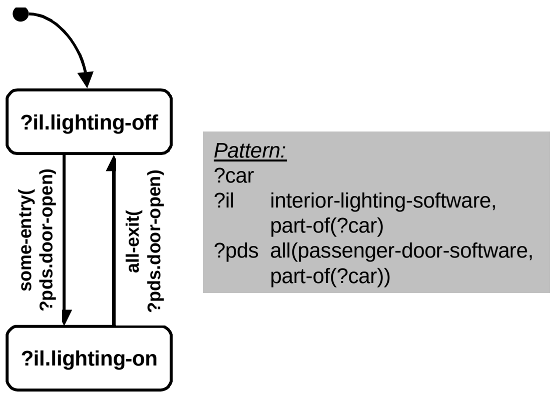

Figure 1. Example of a statechart

3.2 Behavior models in the taxonomical structure

Let us consider the specialization of concept C to concept C'. Each property of C' – also the behavior – must potentially be a property of C. For behavior that means that the set of C' traces must be a subset of the C traces (this also corresponds to the comprehension of behavioral inheritance in [11] and [12]). The superconcept's more extensive set of traces means a greater scope of behavior alternatives, whereas specialized objects have less alternatives for their behavior as they are described more specifically. Figure 2 shows the specialization of behavior of an example concept. For building up the taxonomical hierarchy it is enough to model the behavior of the domain objects on the lowest specialization levels. All higher levels can be automatically produced, for example by unifying the behavior traces (or unifying the finite state machines corresponding to the statecharts, resp., depending on what internal representation is used; see above).

3.3 Behavior models in the compositional structure

As an aggregate A is composed from components (e.g. from C1 and C2), the corresponding statechart can also be composed from the components' statecharts. Considering the traces we can build combinations of the states in the aggregate's components up to new states in the aggregate's traces, as can be seen in figure 3. Although in the example the statechart corresponding to A consists of two parts graphically, it is a closed model whose parts are connected with each other by the entry conditions.

Decomposing an object into its components permits a high number of options in general, as not all potential components of an object (or a module resp.) need be choosen (and instantiated) in a given configuration. A decomposition is only permissible if the restrictions between the partial components' statecharts, which are given by domain patterns, are fulfilled. In the example (fig. 3) the pattern variables *?x* and *?y* of the component C2 can be mapped onto the states *c* and *a* of the component C1.

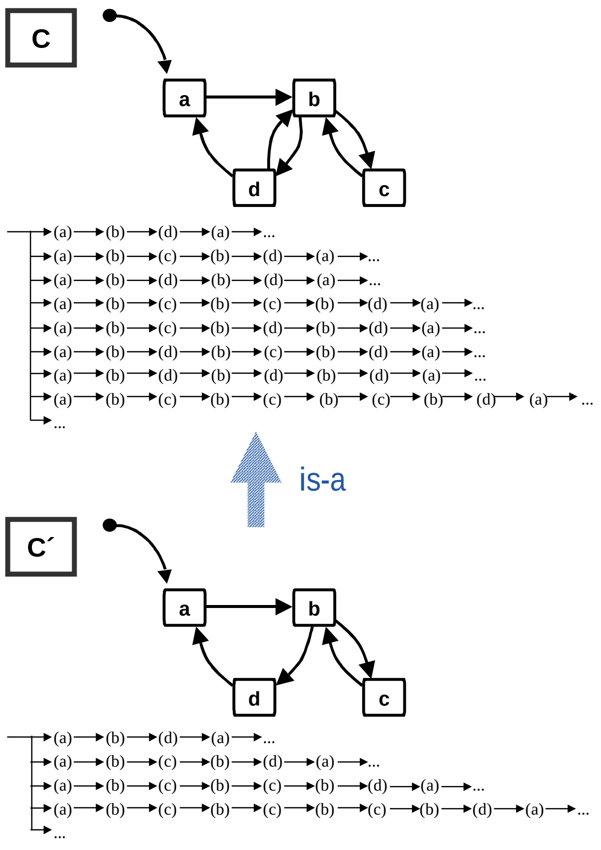
C
a
b
c
d
(a)→(b)→(d)→(a)→...
(a)→(b)→(c)→(b)→(d)→(a)→...
(a)→(b)→(d)→(b)→(d)→(a)→...
(a)→(b)→(c)→(b)→(c)→(b)→(d)→(a)→...
(a)→(b)→(c)→(b)→(d)→(b)→(d)→(a)→...
(a)→(b)→(d)→(b)→(c)→(b)→(d)→(a)→...
(a)→(b)→(d)→(b)→(d)→(b)→(d)→(a)→...
(a)→(b)→(c)→(b)→(c)→(b)→(c)→(b)→(d)→(a)→...
...
is-a
C´
a
b
c
d
(a)→(b)→(d)→(a)→...
(a)→(b)→(c)→(b)→(d)→(a)→...
(a)→(b)→(c)→(b)→(c)→(b)→(d)→(a)→...
(a)→(b)→(c)→(b)→(c)→(b)→(c)→(b)→(d)→(a)→...
...

Figure 2. Specialization of a concept's behavior, described by statecharts and state traces

In the following section it will be described how the behavior models can be used to draw conclusions during the configuration process.

4 CONFIGURATION PROCESS USING BEHAVIOR KNOWLEDGE

Both the structural knowledge (arrangement of domain objects in a taxonomic hierarchy with relationships and parameters) and the behavior knowledge (state-based behavioral description as statecharts or any other internal representation) can serve as a starting point for configuration decisions. In this context, behavior-based decision-making should not compete with structure-based decision-making, rather they should complement one another.

Also the specification of a configuration goal can comprise of both structural and behavioral requirements. This means that when the configuration objectives are entered, the target object and required properties[3] can be specified just as well as particular requirements for the behavior of the system to be configured, which in our approach can be specified by (incomplete) statecharts. The total behavior should not have to be stipulated by the user (just as little as the whole structure of the system), this is "configured" in the solution process.

Whilst we will not be going further into how knowledge structure is utilized for configuration decisions (for this see e.g. [7], [10]), it will be described how the behavior knowledge can be used to perform the four types of configuration steps given in section 2 (specialization, decomposition, integration and parameterization). Correspondingly it could be possible to extend the control of the PLAKON, KONWERK or EngCon systems (see [4], [6], [1]).

Specializing an instance. The specialization of a software module instance means to transmute this instance into a more specific module, i.e. (among other effects) to reduce the potential value ranges of its properties. For the behavior this means that the set of possible traces will be reduced to a subset (see above). In this way, the behavior model description can be transmuted – by transmuting the instance into a more specific subinstance – into a specialized behavior, which is suitable for the behavior demanded in the task specification. To ensure this, the specialized behavior has to be checked against the statechart in the task specification.

Decomposing an instance. While a module instance is decomposed, its accessory submodules could be instantiated and become elements of the current solution. As well as the modules being decomposed, the behavior models belonging to these modules are also decomposed. Concomitant to the decomposition and instantiation of components, relating the participated statecharts to each other can be performed on the basis of bindings in the domain pattern. That is to say, variables in the domain pattern can be bound to the respective instance. This enables the accessing of respective instances in the statecharts' conditions.

Integrating an instance. Integrating a component into an aggregate means exchanging a generic subconcept in the aggregate by a specific instance of this concept. As said before the behavior of the instance might be more precise than the concept's behavior (as the set of traces might be reduced). It has to be guaranteed that this specialized behavior still suits the other subconcepts or instances of the aggregate. If this is not given, the configuration control will not permit this integration step. As with integration, an existing instance (reducing its concept's scope) is assigned to an aggregate, the integration can also be seen as a kind of specialization of the aggregate.

[3] See for example [18].

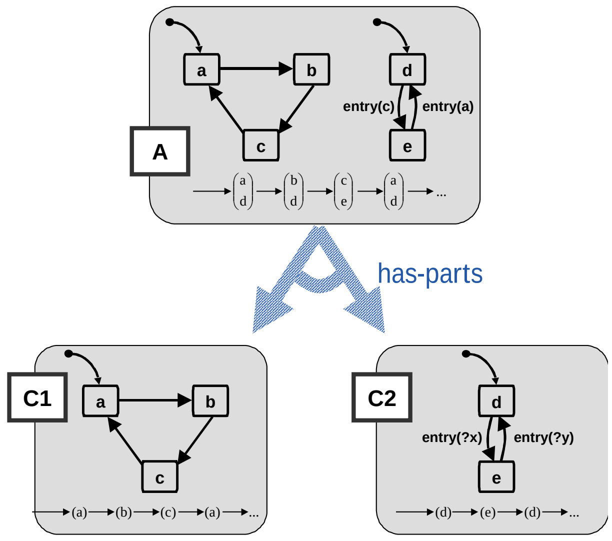

Figure 3. Consideration of behavior during aggregation of C1 and C2 to A

Parameterization of an instance. Fundamentally, parameters in the behavior description can be handled similarly to the remaining parameters (properties) of an instance, i.e. parameterization also takes place here by a reduction of the range of values for the behavior parameters.

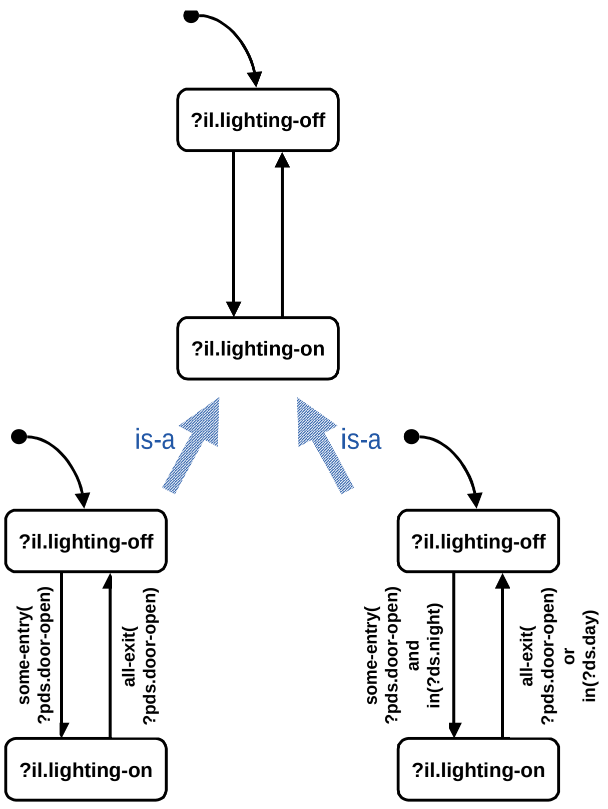
?il.lighting-off
?il.lighting-on
is-a
is-a
?il.lighting-off
some-entry(?pds.door-open)
all-exit(?pds.door-open)
?il.lighting-on
?il.lighting-off
some-entry(?pds.door-open) and in(?ds.night)
all-exit(?pds.door-open) or in(?ds.day)
?il.lighting-on

Figure 4. State descriptions of the software module *interior lighting* and its specializations

The configuration process, consisting of specialization, decomposition, integration and parameterization steps – as well as further techniques such as constraint propagation, which are not described further here – should result in a set of instances, where each can have behavior. This behavior has to be unequivocal for the configuration solution. As the behavior of all instances are based on the same set of states, all behavior models can be seen as one behavior model. This total behavior must fulfill all behavioral requirements given in the task specification by the user.

5 APPLICATION EXAMPLE: CONFIGURATION OF SOFTWARE-BASED VEHICLE ELECTRONIC SYSTEMS

In the field of current vehicle electronics, the trend of increasing the functionality using software can be noted. The introduction of software provides the possibility of using universal control units in the future, instead of specialized control systems for different functions (e.g. control systems for engines, gears, suspension, etc.). These can

- be programmed easily.
- be modified later.
- carry out several sub functions at the same time.

By giving different vehicles individual software configurations, higher diversity and more customer-appropriate vehicle construction can be achieved. This trend leads to increased demands on the development of the electronic system within the vehicle. On one hand there will be an immense variety of possible variants, whereas on the other, the variants will be subject to increasing change through the simple realization of software updates.

The knowledge-based software configuration approach based on the above described software modeling, gives the ability to support the creation of the high number of vehicle software variants in vehicle development as well as in sales. In sales, the knowledge-based configuration can help to realize individual customer-desired functions, by using an appropriate software configuration for each car. To make this possible, such a configuration has to be based exclusively on hardware and software components which are permitted for the respective vehicle series. Particularly in sales, individual programming of control units performed by engineers and specialists should be avoided. The task rather concerns the composition of the vehicle software from single modules and their parameterization. An example for a behavior model of a software module *interior lighting*, which can be specialized to a *daylight-independent interior lighting* and a *daylight-dependent interior lighting*, is shown in figure 4.

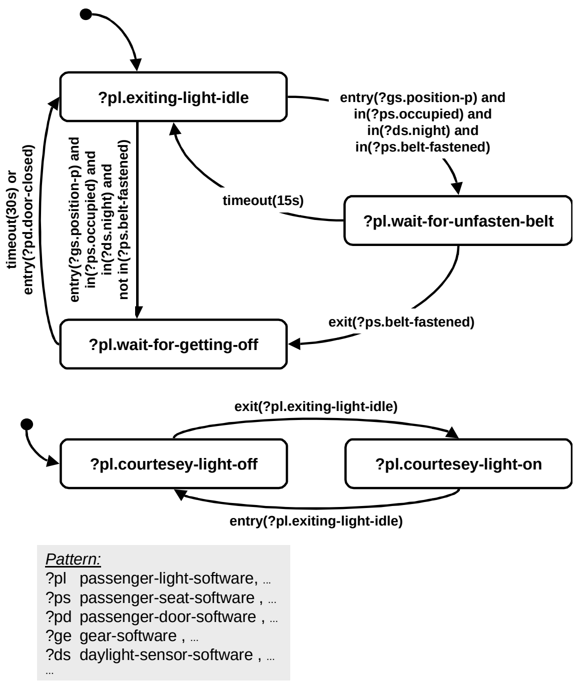

Figure 5. Statechart description of an individual customer demand for the courtesy light

A scenario by way of example: Let's say a customer (e.g. a taxi driver) wants his car to be equipped with a particular courtesy light, that helps the passenger to locate the seat belt lock and the handle. This should be achieved by the automatic illumination of the interior lighting on the passenger's side as soon as the driver switches the automatic gear stick to "P" (parking). If the passenger does not leave the car within a predefined period, the light should switch off automatically.

A sales employee could record such a customer demand by means of a (not necessarily complete) statechart, which could serve as a task specification for the configuration system (see figure 5). The configuration system's task is now to determine the software modules to be used, to parameterize them, to distribute them onto the existing control units and to determine their sequences. This includes the determination of the bus communication for each control unit.

6 SUMMARY AND OUTLOOK

Configuring complex software-based systems, which are able to carry out reactive behavior – e.g. the software-based electronic systems of vehicles in the future – can be a very intricate task. Configuration techniques that exceed current configuration approaches are required. The system behavior should especially be taken into account during configuration. We suggest a modeling approach for knowledge on software, which is based on a software module concept hierarchy, implying taxonomical, compositional and interface relationships. Beside other module properties, each module concept can be given a description of the module behavior on an abstract level, e.g. using statecharts. These behavior descriptions can be used for drawing configuration decisions such as specializing, decomposing, integrating or parameterization of module concepts. As the individual configuration steps can be based on this behavior knowledge, our approach extends to current structure-based configuration methodologies.

In opposition to simulation-based configuration approaches our method allows performing of the basic configuration steps (specializing, composing, integrating or parameterization) directly based on the behavior model, whereas when using simulation, configuration decisions first have to be driven (using a heuristics) and afterwards the results can be evaluated by simulation.

This paper is confined to the basic types of configuration steps. There are further techniques, especially constraint propagation, which promise to have favorable effects if they were also extended to use behavioral configuration knowledge. By way of contrast to the behavior descriptions given here, which are each assigned to the domain objects (and always assigned to *a particular* domain object), constraints represent restrictions between several domain objects. In this way they should also be able to describe *behavioral* restrictions between domain objects. A corresponding extension of the current constraint techniques for configuration, which so far disregard temporal (and therefore behavioral) dependencies is needed. This topic may be dealt with in the future.

For the future, the implementation of a prototypical configuration system is planned, which comprises the methods outlined in this article. It shall be used for configuring vehicle software systems to implement customer-individual vehicle functionality.

REFERENCES

[1] Arlt, V.; Günter, A.; Hollmann, O.; Wagner, T.; Hotz, L. *EngCon – Engineering & Configuration*. In Friedrich, G. ed. *Configuration – Papers from the AAAI Workshop (Orlando, Florida)*, Technical Report WS-99-05, AAAI Press, California, 1999

[2] Biundo, S.; Günter, A.; Hertzberg, J.; Schneeberger, J.; Tank, W. *Planen und Konfigurieren*. In Görz, G. ed. *Einführung in die künstliche Intelligenz*, 2nd edition, Addison-Wesley Publishing, Berlin, 1995

[3] Bradshaw, J.; Young, R. M. *Evaluating Design Using Knowledge of Purpose and Knowledge of Structure*. In IEEE Expert, Vol. 6 (2), pp. 33-40, 1991

[4] Cunis, R., Günter, A., Strecker, H. ed. *Das PLAKON Buch – Ein Expertensystemkern für Planungs- und Konfigurierungsaufgaben in technischen Domänen*, Springer, Berlin, 1991

[5] Douglass, B. P. *Real-Time UML – Developing Efficient Objects for Embedded Systems*, Addison Wesley, Reading, Massachusetts, 1998

[6] Günter, A. KONWERK – ein modulares Konfigurierungs-werkzeug. In Maurer, F.; Richter, M. M. eds. *Expertensysteme 95*, Kaiserslautern, infix Verlag, pp. 1-18, 1995

[7] Günter, A. ed. *Wissensbasiertes Konfigurieren – Ergebnisse aus dem Projekt PROKON*. St. Augustin, Germany: infix Verlag, 1995

[8] Günter, A.; Hotz, L. *KONWERK – A Domain Independent Configuration Tool*. In Friedrich, G. ed. *Configuration – Papers from the AAAI Workshop (Orlando, Florida)*, Technical Report WS-99-05, AAAI Press, California, 1999

[9] Günter, A.; Kühn, C. *Einsatz der Simulation zur Unterstützung der Konfigurierung von technischen Systemen*. In Mertens, P.; Voss, H. eds. *Expertensysteme 97 – Beiträge zur 4. Deutschen Tagung Wissensbasierte Systeme (XPS-97)*, Bad Honnef am Rhein, infix Verlag, pp. 93-106, 1997

[10] Günter, A.; Kühn, C. *Knowledge-Based Configuration – Survey and Future Directions*. In Puppe, F. ed. *XPS-99: Knowledge Based Systems, Proceedings 5th Biannual German Conference on Knowledge Based Systems*, Springer Lecture Notes in Artificial Intelligence 1570, Germany, 1999

[11] Harel, D.; Gery, E. *Executable Object Modeling with Statecharts*. IEEE Computer, Vol. 30, No. 7 (July), pp. 31-42, 1997

[12] Harel, D.; Kupferman, O. *On the Inheritance of State-Based Object Behavior*. To appear, 2000

[13] Harel, D.; Politi, M. *Modeling Reactive Systems with Statecharts*. McGraw-Hill, 1998

[14] Kühn, C. *Requirements for Configuring Complex Software-Based Systems*. In Friedrich, G. ed. *Configuration – Papers from the AAAI Workshop (Orlando, Florida)*, Technical Report WS-99-05, AAAI Press, California, 1999

[15] Kühn, C.; Günter, A. *Combining Knowledge-Based Configuration and Simulation*. In Kleine Büning, H. ed. *Beiträge zum Workshop „Simulation in Wissensbasierten Systemen“ (SiWiS-98)*, report tr-ri-98-194, Universität-GH Paderborn, Germany, 1998

[16] Stein, B. *Functional Models in Configuration Systems*. Dissertation, University of Paderborn (GH), 1995

[17] Stumptner, M. *An overview of knowledge-based configuration*. AI Com 10 (2), pp. 111-126, 1997

[18] Thärigen, M. *Wissensbasierte Erfassung von Anforderungen*. In Günter, A. ed. *Wissensbasiertes Konfigurieren – Ergebnisse aus dem Projekt PROKON*, 89-96. St. Augustin, Germany: infix Verlag, 1995