From Model-Based Design to Formal Verification of Adaptive Embedded Systems*

Rasmus Adler[1], Ina Schaefer[2], Tobias Schuele[3], and Eric Vecchié[3]

[1] Fraunhofer Institute for Experimental Software Engineering (IESE), Kaiserslautern, Germany
Rasmus.Adler@iese.fraunhofer.de
[2] Software Technology Group, Department of Computer Science, University of Kaiserslautern, Germany
inschaef@informatik.uni-kl.de
[3] Reactive Systems Group, Department of Computer Science, University of Kaiserslautern, Germany
{schuele,vecchie}@informatik.uni-kl.de

Abstract. Adaptation is important in dependable embedded systems to cope with changing environmental conditions. However, adaptation significantly complicates system design and poses new challenges to system correctness. We propose an integrated model-based development approach facilitating intuitive modelling as well as formal verification of dynamic adaptation behaviour. Our modelling concepts ease the specification of adaptation behaviour and improve the design of adaptive embedded systems by hiding the increased complexity from the developer. Based on a formal framework for representing adaptation behaviour, our approach allows to employ theorem proving, model checking as well as specialised verification techniques to prove properties characteristic for adaptive systems such as stability.

1 Introduction

Many embedded systems autonomously adapt at runtime to changing environmental conditions by up- and downgrading their functionality according to the current situation. Adaptation is particularly important in safety-critical areas such as the automotive domain to meet the high demands on dependability and fault-tolerance. For this reason, adaptation has become state-of-the-art in antilock braking, vehicle stability control and adaptive cruise control systems. For example, if the sensor measuring the yaw rate of a car fails, the vehicle stability control system may adapt to a configuration, where the yaw rate is approximated by steering angle and vehicle speed. In this way, it can be guaranteed that the system is still operational even if some of the components fail in order to provide a maximum degree of safety and reliability. However, adaptation significantly complicates the development of embedded systems. One reason for

* This work has been supported by the Rheinland-Pfalz Cluster of Excellence 'Dependable Adaptive Systems and Mathematical Modelling' (DASMOD).

M. Butler, M. Hinchey, M.M. Larrondo-Petrie (Eds.): ICFEM 2007, LNCS 4789, pp. 76–95, 2007.

this is that in the worst case the number of configurations a system can adapt to is exponential in the number of its modules. Moreover, for ensuring system correctness it is not sufficient to consider each configuration separately but the adaptation process as a whole has to be checked.
A promising approach to deal with the increased complexity posed by adaptation is model-based design. As a major advantage, model-based design allows to focus on the needs of each phase in the design process and to model the required concepts as close as possible to the intuition by capturing them in an accurate and understable manner. Regarding the development of adaptive systems, model-based design supports the validation and verification of adaptation behaviour before the actual funtionality is implemented. The integration of formal verification into the development process is important to rigorously prove that the adaptation behaviour meets critical requirements such as stability.

In this paper, we propose an integrated framework for model-based design and formal verification of adaptive embedded systems. The modelling concepts of our approach hide the complexity at system level by fostering modular design and independent specification of functionality and adaptation behaviour. In this way, the designer can concentrate on the adaptation behaviour during early phases of the design process without having to consider implementation specific details. The design can then be refined successively by adding the intended functionality.

In order to formally reason about adaptive embedded systems, we propose a framework that captures the semantics of the modelling concepts at a high level of abstraction. Using this framework, the models as well as the desired properties can be formulated in a semantically exact manner. This is particularly important regarding the application of different verification techniques: Firstly, it is possible to embed the models into a representation suitable for a theorem prover and to verify the specified properties directly, e.g. by means of induction. Secondly, properties frequently occurring in the verification of adaptive system can be checked by automatic techniques such as symbolic model checking.

However, many systems encountered in practice are not directly amenable to formal verification by model checking due to their huge state space. To solve this problem, our formal framework allows to perform transformations on the models in order to reduce verification complexity. For example, data abstraction techniques may be employed to reduce the state space. The separation between functionality and adaptation behaviour is thereby maintained, which allows to consider purely functional, purely adapative and combined aspects. As the models in our framework have a clear semantics, it can be guaranteed by means of a theorem prover that the applied transformations are property preserving.

For certain properties, it is often advantageous to apply specialised verification methods, as standard model checking procedures are not always as efficient as possible. This is the case for stability of the adaptation process, one of the most important properties in adaptive systems, as adaptations in one component may trigger further adaptations in other components, which may lead to unstable configurations. However, in embedded systems, which are usually subject to certain real-time constraints, it must be guaranteed that a system stabilises after

a bounded number of adaptation steps. To this end, we propose an approach that allows to verify stability of adaptive systems more efficiently than using standard model checking procedures.

To illustrate our approach, we use a building automation system as running example. The system consists of four modules: an occupancy detection, a light control, a lamp and an alarm system. The functionality is as follows: The light in a room is controlled according to the room occupancy. If the room is unoccupied, the lamp is switched off. Otherwise, the lamp is adjusted according to the current illuminance of the room. Additionally, an alarm is raised if the room is occupied without authorisation. Each module has a number of configurations for maintaining its functionality in case of failures. For instance, the module *OccupancyDetection* uses data from a camera, a motion detector, and transponders to determine occupancy of the room. When the camera is defect, the module adapts from camera-based to motion-based occupancy detection.

The rest of this paper is structured as follows: In Section 2, we introduce the concepts for modelling adaptive embedded systems and present the underlying formal framework. In Section 3, we address some aspects of adaptive system verification with a focus on stability. In Section 4, we describe the implementation of our approach. Finally, we discuss related work (Section 5) and conclude with an outlook to future work (Section 6).

2 Modelling Adaptive Embedded Systems

2.1 Concepts for Modelling Adaptation Behaviour

The objective of our modelling concepts called MARS (Methodologies and Architectures for Runtime Adaptive Systems) is the explicit modelling of adaptation behaviour, which is a prerequisite for its validation and verification. These concepts have been successfully applied in industry and academia for several years and provide a seamlessly integrated approach for the development of adaptive systems [21]. The major difficulty in modelling adaptation behaviour are complex interdependencies between the modules of a system. To solve this problem, we employ the concept 'separation of concerns' by separating functional from adaptation behaviour and the concept 'divide and conquer' by defining the adaptation behaviour modularly within the modules. Based on these concepts, it is possible to hide the complexity at system level from the developer.

A system consists of a set of modules that communicate with each other by passing signals via ports. This is a common notion found in various modelling languages and complies with the definition of architecture description languages by Taylor et al. [10]. In contrast to non-adaptive modules, our modules have several functional behaviours in order to support different degradation levels.

Quality Descriptions. A modular definition of adaptation behaviour is indispensible for handling the enormous complexity of most systems. For this reason, we establish a quality flow in the system making such modular definitions possible. Besides the actual data, each signal has an additional quality description. To

this end, signals are typed by datives (extended data type for adaptive systems) that do not only describe which data values a signal may take, but also how the quality of this data can be described. Hence, a dative consists of a data type and a quality type. The former describes the type of data values like integers or real numbers. The quality type provides type-specific quality information, because a general purpose quality information like the relative error is not reasonable in many cases, e.g. for Boolean signals. Since the quality is part of the type definition, module designers are able to define the adaptation behaviour solely on the basis of quality descriptions available at a module's local interface. Additionally, they define how the current quality of the provided signals is determined.

In order to define the quality of a functional value of a signal, it is necessary to know which behavioural variant has been used to determine a value. In the first place, a quality type is defined by a set of possible modes. A developer using a signal knows the deficiencies associated with a certain mode and decides how a module must adapt in order to compensate for these deficiencies. Additionally, mode attributes can be used to describe the signal quality more precisely using mode-specific characteristics. Consequently, a mode is described by the mode itself and a set of mode attributes.

As an example, Figure 1 shows the definition of the dative *occupancy*. Its quality type contains five modes: The mode *camera* refers to a camera-based occupancy detection and mode *motion* indicates that the occupancy is derived from the detected motions. A deficiency associated with mode *motion* is that only movements are detected instead of actual persons in the room. As the quality of motion-based occupancy detection strongly depends on the reaction point of the motion sensor, the mode attribute *reactionpoint* is attached to the mode *motion*. In the example, *reactionpoint* represents the sensitivity of the motion sensor.

Modules. Based on datives, developers can modularly define the adaptation behaviour of single modules using two extensions made to conventional modules.

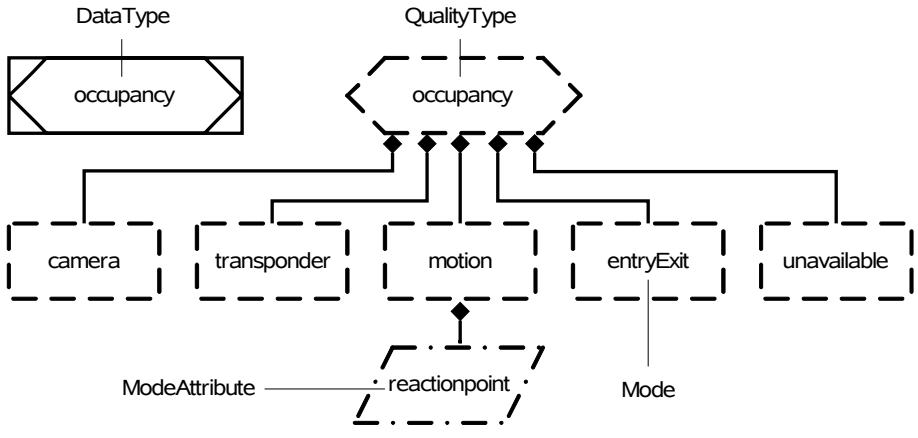

Fig. 1. Example for the definition of a dative

First, the behaviour specification is not directly assigned. Several configurations can be assigned to a module, each of them representing one behaviour variant. Second, in addition to the input/output interface, we define a required/provided interface. This distinction is used for describing the direction of the quality flow. This is not always identical to the direction of the data flow between two connected module ports. Although the connection is typed by one dative, the data part of the dative flows from an output port of one module to an input port of another module, while the quality flows from a provided port to a required port. This can for instance be the case for an actuator where a data value is propagated to the actuator while the actuator's status is conveyed to the functional unit via the signal's quality. The interface of a module is defined by a set of input signals, a set of output signals, a set of required signals and a set of provided signals.

Configurations. A module can be in one of several configurations, each of them representing one behavioural variant. A module is thus defined by its interface and a set of configurations. In our running example, the module *OccupancyDetection* can be in one of five configurations, depending on how occupancy of a room is determined. For instance, *CameraDetection* is the configuration, where the occupancy is derived from a camera image. A configuration is defined by the following elements: (1) a specification of the associated behavioural variant, (2) a guard defining under which conditions the configuration can be activated, (3) a priority and (4) an influence defining how the quality of the provided signals is determined.

A guard is a Boolean expression. If the guard evaluates to true at run time, the configuration can be activated. Operands of guards are quality descriptions of required signals. A guard defines which signals are required in which mode and which values the mode attributes may have. For instance, the guard of the configuration *MotionDetection* in module *OccupancyDetection* defines that the required quality of the signal *detected_motion* has to be in mode 'available'. Additionally, it could be enforced that the mode attribute *reactionpoint* is in a certain range. Often, guards of several configurations are satisfied at the same time. Therefore, an unambiguous priority is assigned to each configuration. At run time, the configuration with the highest priority is activated and the associated behaviour is executed. Influence rules describe how the quality of the provided signals is determined. Each influence rule consists of an influence guard and an influence function. The influence guard refines the configuration guard and defines a condition under which the respective influence function is applied. The influence function assigns the appropriate mode to each provided signal and calculates the mode attributes. For instance, configuration *MotionDetection* has only one influence rule whose influence function assigns the quality of signal *occupancy* to mode *motion*.

2.2 Formal Representation of Modelling Concepts

In this subsection, we show how the modelling concepts of MARS can be formally represented by Synchronous Adaptive Systems (SAS), which constitute the

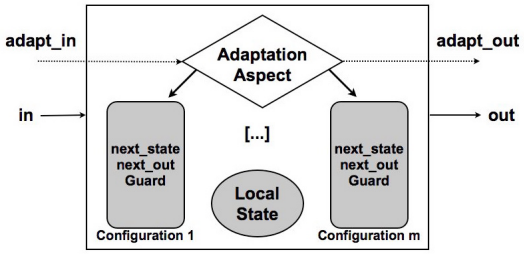

Fig. 2. Separation of functional and adaptation behaviour in an SAS module

basis for formal verification of adaptive embedded systems [16]. SAS capture the semantics of adaptation behaviour at a high level of abstraction bridging the gap between the modelling concepts and their formal representation. The modularity provided by MARS is represented by composing synchronous adaptive systems from a set of modules. Each module comprises a set of predetermined behavioural configurations it may adapt to. SAS maintain the separation of adaptive and functional behaviour. This is accomplished by defining an adaptation aspect on top of the different functional configurations. The active configuration is determined by the adaptation aspect. SAS are assumed to be open systems with input provided by the environment. Furthermore, they are modelled synchronously as their simultaneously invoked actions are executed in true concurrency. Figure 2 depicts the intuitive notion of a module.

For the definition of SAS syntax, we assume a set of distinct variable names *Var* and a set of values *Val* that can be assigned to these variables. The formal definition of modules is based on state transition systems.

Definition 1 (Module and Adaptation). *An SAS module m is a tuple $m = (in, out, loc, init, confs, adaptation)$ with*

- *$in \subseteq Var$, the set of input variables, $out \subseteq Var$, the set of output variables, $loc \subseteq Var$, the set of local variables and $init : loc \rightarrow Val$ their initial values*
- *$confs = \{(guard_j, next\_state_j, next\_out_j) \mid j = 1, ..., n\}$ the configurations of the module, where*
  - *$guard_j$: the Boolean closure of constraints on $\{adapt\_in, adapt\_loc\}$ determining when configuration j is enabled with $adapt\_in$ and $adapt\_loc$ as defined below*
  - *$next\_state_j$: $(in \cup loc \rightarrow Val) \rightarrow (loc \rightarrow Val)$ the next state function for configuration j*
  - *$next\_out_j$: $(in \cup loc \rightarrow Val) \rightarrow (out \rightarrow Val)$ the output function for configuration j*

The adaptation aspect is defined as a tuple $adaptation = (adapt\_in, adapt\_out, adapt\_loc, adapt\_init, adapt\_next\_state, adapt\_next\_out)$, where

- *$adapt\_in \subseteq Var$ is the set of adaptation in-variables, $adapt\_out \subseteq Var$ the set of adaptation out-variables, $adapt\_loc \subseteq Var$ the set of adaptation local state variables and $adapt\_init : adapt\_loc \rightarrow Val$ their initial values*

- $adapt\_next\_state : (adapt\_in \cup adapt\_loc \rightarrow Val) \rightarrow (adapt\_loc \rightarrow Val)$ *the adaptation next state function*
- $adapt\_next\_out : (adapt\_in \cup adapt\_loc \rightarrow Val) \rightarrow (adapt\_out \rightarrow Val)$ *the adaptation output function*

The module concept of MARS is represented by SAS modules where module ports are mapped to input and output variables. The dative associated with a port is modelled by a set of variables: one functional variable for the functional data, an adaptive variable for each mode and additional adaptive variables for mode attributes. In the running example, the module *OccupancyDetection* is represented by an SAS module. The input signal *motion_detected* is split into two variables, a functional variable *motion_detected* and an adaptive variable *motion_detected_quality* carrying the mode of the signal. A configuration in a module is represented by an SAS configuration, where the configuration guard is mapped to an SAS configuration guard and the priority to the configuration index. The configuration behaviour is expressed by the *next_output* function of the configuration. So, the configuration *MotionDetection* in module *Occupancy-Detection* is represented by an SAS configuration with a guard expressing that the adaptive variable *motion_detected_quality* must have the value 'available'. The influence function of a configuration is represented using the *adapt_next_out* function of the SAS module's adaptation aspect. In our example the adaptive output variable *occupancy_quality* corresponding to the quality part of the signal *occupancy* is assigned to the mode *motion* by the *adapt_next_out* function if the configuration *MotionDetection* is used. Since MARS concepts currently do not use state variables, the respective parts of SAS remain unused.

An SAS is composed from a set of modules that are interconnected via their own and the system's input and output variables. For technical reasons, we assume that all system variable names and all module variable names are disjoint. Whereas for module ports in MARS it is defined whether a quality is required or provided, quality and data flow in SAS are completely decoupled using separate adaptive connections. Hence, provided ports are mapped to adaptation output variables and required ports are mapped to adaptation input variables. A module can trigger adaptations in other modules via adaptive connections.

Definition 2 (SAS). *A synchronous adaptive system S is a tuple*

$$S = (M, input_a, input_d, output_a, output_d, conn_a, conn_d),$$

where

- $M = \{m_1, \ldots, m_n\}$ *is a set of SAS modules with* $m_i = (in_i, out_i, loc_i, init_i, confs_i, adaptation_i)$
- $input_a \subseteq Var$ *are adaptation inputs and* $input_d \subseteq Var$ *functional inputs to the system*
- $output_a \subseteq Var$ *are adaptation outputs and* $output_d \subseteq Var$ *functional outputs from the system*

- *$conn_a$ is a function connecting adaptation outputs to adaptation inputs, system adaptation inputs to module adaptation inputs and module adaptation outputs to system adaptation outputs, i.e. $conn_a : \bigcup_{j,k=1,\ldots,n}(adapt\_out_j \cup input_a) \rightarrow (adapt\_in_k \cup output_a)$, where $conn_a(input_a) \subseteq adapt\_in_k$*
- *$conn_d$ is a function connecting outputs of modules to inputs, system inputs to module inputs and module outputs to system outputs, i.e. $conn_d : \bigcup_{j,k=1,\ldots,n}(out_j \cup input_d) \rightarrow (in_k \cup output_d)$, where $conn_d(input_d) \subseteq in_k$.*

The semantics of SAS is defined in a two-layered approach. We start by defining the local semantics of single modules similar to standard state-transition systems. From this, we define global system semantics. A local state of a module is defined by a valuation of the module's variables, i.e. input, output and local variables and their adaptive counterparts. A local state is initial if its functional and adaptation variables are set to their initial values and input and output variables are undefined. A local transition between two local states evolves in two stages: First, the adaptation aspect computes the new adaptation local state and the new adaptation output from the current adaptation input and the previous adaptation state. The adaptation aspect further selects the configuration with the smallest index that has a valid guard with respect to the current input and the previous functional and adaptation state. The system designer should ensure that the system has a built-in default configuration 'off' which becomes applicable when no other configuration is. The selected configuration is used to compute the new local state and the new output from the current functional input and the previous functional state.

Definition 3 (Local States and Transitions). *A local state s of an SAS module m is a variable assignment:*

$$s : in \cup out \cup loc \cup adapt\_in \cup adapt\_out \cup adapt\_loc \rightarrow Val$$

A local state s is called initial iff $s|_{loc} = init$, $s|_{adapt\_loc} = adapt\_init$ and $s|_V = undef$ for $V = in \cup out \cup adapt\_in \cup adapt\_out$.[1] *An SAS module performs a local transition between two local states s and s', written $s \rightsquigarrow s'$, iff the following conditions hold:*

$$\begin{gathered}
s'|_{adapt_loc} = adapt_next_state(s'|_{adapt_in} \cup s|_{adapt_loc}) \\
s'|_{adapt_out} = adapt_next_out(s'|_{adapt_in} \cup s|_{adapt_loc}) \\
\forall\ 0 < j < i\ .\ s'|_{in} \cup s|_{loc} \cup s'|_{adapt_in} \cup s|_{adapt_loc} \not\models guard_j \\
s'|_{in} \cup s|_{loc} \cup s'|_{adapt_in} \cup s|_{adapt_loc} \models guard_i \\
s'|_{loc} = next_state_i(s'|_{in} \cup s|_{loc}) \text{ and } s'|_{out} = next_out_i(s'|_{in} \cup s|_{loc})
\end{gathered}$$

The state of an SAS is the union of the local states of the contained modules together with an evaluation of the system inputs and outputs. A system state is initial if all states of the contained modules are initial and the system input and output is undefined. A transition between two global states is performed in

[1] For a function f and a set M, $f|_M = \{(x, f(x)) \mid x \in M\}$ is the restriction of f to the domain M.

three stages. Firstly, each module reads its input either from another module's output of the previous cycle or from the system inputs in the current cycle. Secondly, each module synchronously performs a local transition. Thirdly, the modules directly connected to system outputs write their results to the output variables.

Definition 4 (Global States and Transitions). *A global state σ of an SAS consists of the local states $\{s_1, \ldots, s_n\}$ of the contained modules, where s_i is the state of $m_i \in M$, and an evaluation of the functional and adaptive inputs and outputs, i.e. $\sigma = s_1 \cup \ldots \cup s_n \cup ((input_a \cup input_d \cup output_a \cup output_d) \rightarrow Val)$. A global state σ is called initial iff all local states s_i for $i = 1, \ldots, n$ are initial and the system inputs and outputs are undefined. Two states σ and σ' perform a global transition, written $\sigma \rightarrow_{glob} \sigma'$, iff*

- *for all $x, y \in Var \setminus (input_d \cup input_a)$ with $conn_d(x) = y$ or $conn_a(x) = y$ it holds that $\sigma'(y) = \sigma(x)$, for all $x \in input_a$ and $y \in Var$ with $conn_a(x) = y$ it holds that $\sigma'(y) = \sigma'(x)$ and for all $x \in input_d$ and $y \in Var$ with $conn_d(x) = y$ it holds that $\sigma'(y) = \sigma'(x)$*
- *for all $s_j \in \sigma$ and for all $s'_j \in \sigma'$ it holds that $s_j \rightsquigarrow s'_j$*
- *for all $x \in Var$ and $y \in output_d$ with $conn_d(x) = y$ it holds that $\sigma'(y) = \sigma'(x)$ and for all $x \in Var$ and $y \in output_a$ with $conn_a(x) = y$ it holds that $\sigma'(y) = \sigma'(x)$*

A sequence of global states $\sigma^0\sigma^1\sigma^2 \ldots$ of an SAS is a path if σ^0 is an initial global state and for all $i \geq 0$ we have $\sigma^i \rightarrow_{glob} \sigma^{i+1}$. The set $Paths(SAS) = \{\sigma^0\sigma^1\sigma^2 \ldots \mid \sigma^0\sigma^1\sigma^2 \ldots \text{ is a path}\}$ constitutes the SAS semantics.

3 Verification

The properties to be verified for adaptive embedded systems can be classified according to whether they refer to adaptive, functional or both aspects. Moreover, one can distinguish between generic properties that are largely independent of the application and application specific properties. In the following, we will concentrate on generic properties of the adaptation behaviour.

As specification languages, we use the temporal logics CTL (computation tree logic) and LTL (linear time temporal logic) [6,18]. In both CTL and LTL, temporal operators are used to specify properties along a given computation path. For example, the formula $\mathsf{F}\varphi$ states that φ eventually holds and $\mathsf{G}\psi$ states that ψ invariantly holds. In CTL, every temporal operator must be immediately preceeded by one of the path quantifiers A (for all paths) and E (at least one path). Thus, $\mathsf{AG}\varphi$ and $\mathsf{EF}\psi$ are CTL formulae stating that φ invariantly holds on all paths and ψ eventually holds on at least one path, respectively. LTL formulae always have the form $\mathsf{A}\varphi$, where φ does not contain any path quantifiers. None of these two logics is superior to the other, i.e., there are specifications that can be expressed in LTL, but not in CTL, and vice versa. However, both are subsumed by the temporal logic CTL* [6,18].

SAS models can be verified directly by embedding them into a semantic representation of an interactive theorem prover such as Isabelle/HOL [12]. As a major advantage, interactive theorem provers do not suffer from the state explosion problem, as many properties can be verified without having to enumerate all possible states. On the other hand, it is often more convenient to employ automatic verification methods such as model checking, since using a theorem prover can be rather tedious. In the remainder of this section, we will therefore focus on the application of standard and specialised model checking procedures for the verification of SAS models.

3.1 System Transformations

As mentioned in the introduction, SAS models are usually not directly amenable to model checking due to their complexity. Sources of complexity are for instance unbounded data domains, the size of arithmetic constants or the mere size of the model. In order to reduce the runtime of the verification procedures, we perform a number of transformations on SAS models transparent to the user [2]. These transformations are formally verified to be property preserving using Isabelle/HOL.

To deal with unbounded data domains or large constants, we apply the concept of data domain abstraction [5]. Data values from a large or infinite domain are thereby mapped to a smaller finite domain using a homomorphic abstraction function, provided that the domain abstraction is compatible with the operations of the system. Alternatively, one may apply abstract interpretation based techniques [7] that overapproximate the effect of certain operations in the abstracted system and yield a conservative abstraction of the system behaviour. Hence, properties to be verified are abstracted such that an abstracted property implies the original property. As an example, consider a system input ranging over the integers. The integer domain may be reduced to the abstract domain $\{low, high\}$ such that an integer value v is mapped to low iff $v < 50$ and to $high$ iff $v \geq 50$. A constraint on the input like $input \geq 50$ is subsequently transformed to $input = high$ without loosing precision due to the suitably chosen abstraction.

Moreover, we restrict the model to those parts that are relevant for verifying the property under consideration. This means that we first remove all variables that are declared but never used in the model. Furthermore, we perform an analysis which variables of the system model and which associated parts influence the considered property. Unnecessary parts of the model can safely be removed. This technique is known as cone of influence reduction [6] in model checking of Boolean circuits.

SAS models also support reasoning about purely adaptive, purely functional or combined aspects of system models by separating functional from adaptive behaviour. Since model checking tools do in general not have any means to distinguish between functionality and adaptation, the generation of different verification problems from SAS models alleviates verification complexity. For purely adaptive properties, we generate verification output containing only the adaptive part of the models, i.e. adaptive variables and the associated transition functions.

Together with a system transformation, we provide a formal proof that the transformation is property preserving. This means that for a given SAS and a given property, the transformed system satisfies the transformed property if the original property is true in the original system. Our approach is based on translation validation techniques previously applied in compilers. We use a correctness criterion based on property preservation by simulation for the universal fragment of CTL*. We prove in the interactive theorem prover Isabelle/HOL [12] that for each transformation the transformed system simulates the original system and that the transformed property can be concretised to imply the original one. Then, validity of the transformation is established (cf. [2]).

3.2 Verification of Generic Properties by Model Checking

Most of the generic properties can be expressed in CTL, which allows us to employ standard model checking techniques. To verify such properties, we translate the reduced SAS model to the input description of the model checker. First of all, we want to verify that no module gets stuck in the default configuration 'off'. This can be expressed by the CTL formula $\mathsf{AG}(c = \mathit{off} \rightarrow \mathsf{EF}\, c \neq \mathit{off})$, where c stores the current configuration. The next specification is even stronger and asserts that every module can reach all configurations at all times: $\mathsf{AG}(\bigwedge_{i=1}^{n} \mathsf{EF}\, c = \mathit{config}_i)$. If this specification holds, the system is deadlock-free and no configuration is redundant. Moreover, a module must always be in one of the predefined configurations such that no inconsistent states can be reached: $\mathsf{AG}(\bigvee_{i=1}^{n} c = \mathit{config}_i)$.

Many application specific properties can also be verified using standard model checking techniques. On the one hand, these properties are concerned with the adaptation behaviour resulting from the concrete combination of different modules. As an example, one may verify that adaptation in one module leads to a particular configuration in another module after a certain number of cycles. If, for instance, the camera in the building automation system fails, the module *OccupancyDetection* will switch to configuration *MotionDetection* in the next cycle. On the other hand, application specific properties address the functionality of a system. For example, in the building automation scenario, one may verify that the occupancy of the room is determined correctly indepedent of the used configurations and the order of their activation.

3.3 Verification of Stability

As mentioned in the introduction, one of the most important properties of adaptation is stability [15]. Since adaptation in the considered class of systems is not controlled by a central authority, adaptation in one module may trigger further adaptations in other modules. While *finite* sequences of adaptations are usually intended, cyclic dependencies between the modules may lead to an *infinite* number of adaptations, which results in an unstable system. For this reason, it is important to verify that the configurations of a module eventually stabilise if the inputs do not change.

As stability can be expressed in LTL (but not in CTL), it can be verified using standard model checking procedures for LTL. However, model checking

procedures for temporal logic formulae are not always as efficient as specialised verification procedures for certain properties. In particular, there are more efficient ways to check stability, as we will show in this section. Before we go into detail, we briefly describe the μ-calculus, which we will use as the basis of our approach. More detailed information on the μ-calculus can be found in [6,18].

In order to define the syntax and semantics of the μ-calculus, we need the notion of Kripke structures. In our implementation (see Section 4), SAS models are first translated to synchronous programs, which can then be compiled to symbolic descriptions of Kripke structures.

Definition 5 (Kripke structures). *Given a set of variables $\mathcal{V}$, a Kripke structure $\mathcal{K}$ is a labelled transition system $(\mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L})$, where $\mathcal{S}$ is the set of states, $\mathcal{I} \subseteq \mathcal{S}$ is the set of initial states, $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ is the transition relation, and $\mathcal{L}\colon \mathcal{S} \to \mathscr{P}(\mathcal{V})$ is the labelling function that maps each state to a set of variables.*

The predecessors and successors of a set of states are used to define the semantics of the μ-calculus:

Definition 6 (Predecessors and Successors). *Given a Kripke structure $\mathcal{K} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L})$, the predecessors and successors of a set of states $Q \subseteq \mathcal{S}$ are defined as follows:*

- $\mathsf{pre}^{\mathcal{R}}_{\exists}(Q) := \{s \in \mathcal{S} \mid \exists s' \in \mathcal{S}.(s, s') \in \mathcal{R} \wedge s' \in Q\}$
- $\mathsf{pre}^{\mathcal{R}}_{\forall}(Q) := \{s \in \mathcal{S} \mid \forall s' \in \mathcal{S}.(s, s') \in \mathcal{R} \to s' \in Q\}$
- $\mathsf{suc}^{\mathcal{R}}_{\exists}(Q) := \{s' \in \mathcal{S} \mid \exists s \in \mathcal{S}.(s, s') \in \mathcal{R} \wedge s \in Q\}$
- $\mathsf{suc}^{\mathcal{R}}_{\forall}(Q) := \{s' \in \mathcal{S} \mid \forall s \in \mathcal{S}.(s, s') \in \mathcal{R} \to s \in Q\}$

Definition 7 (Syntax of the μ–Calculus). *Given a set of variables $\mathcal{V}$, the set of μ–calculus formulae Form_μ is defined as follows with $x \in \mathcal{V}$ and $\varphi, \psi \in \mathsf{Form}_\mu$:*

$$\mathsf{Form}_\mu := x \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \Diamond\varphi \mid \Box\varphi \mid \overleftarrow{\Diamond}\varphi \mid \overleftarrow{\Box}\varphi \mid \mu x.\varphi \mid \nu x.\varphi$$

Intuitively, a modal formula $\Diamond\varphi$ holds in a state iff φ holds in at least one successor state, and $\Box\varphi$ holds iff φ holds in all successor states. The operators $\overleftarrow{\Diamond}$ and $\overleftarrow{\Box}$ refer to the past (predecessors) instead of to the future (successors). Finally, the operators μ and ν denote least and greatest fixpoints, respectively. In order to define the semantics of the μ-calculus, we denote the subset of states satisfying a formula $\varphi \in \mathsf{Form}_\mu$ by $[\![\varphi]\!]_{\mathcal{K}}$.

Definition 8 (Semantics of the μ–Calculus). *Given a Kripke structure $\mathcal{K} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L})$, the semantics of the μ–calculus is defined as follows, where $\mathcal{K}^Q_x$ is the Kripke structure obtained from $\mathcal{K}$ by changing the states $s \in \mathcal{S}$ such that $x \in \mathcal{L}(s)$ holds iff $s \in Q$ holds:*

$[\![x]\!]_{\mathcal{K}} := \{s \in \mathcal{S} \mid x \in \mathcal{L}(s)\}$ *for all* $x \in \mathcal{V}$

$[\![\varphi \wedge \psi]\!]_{\mathcal{K}} := [\![\varphi]\!]_{\mathcal{K}} \cap [\![\psi]\!]_{\mathcal{K}}$ $\quad$ $[\![\varphi \vee \psi]\!]_{\mathcal{K}} := [\![\varphi]\!]_{\mathcal{K}} \cup [\![\psi]\!]_{\mathcal{K}}$

$[\![\Diamond\varphi]\!]_{\mathcal{K}} := \mathsf{pre}^{\mathcal{R}}_{\exists}([\![\varphi]\!]_{\mathcal{K}})$ $\quad$ $[\![\Box\varphi]\!]_{\mathcal{K}} := \mathsf{pre}^{\mathcal{R}}_{\forall}([\![\varphi]\!]_{\mathcal{K}})$

$[\![\overleftarrow{\Diamond}\varphi]\!]_{\mathcal{K}} := \mathsf{suc}^{\mathcal{R}}_{\exists}([\![\varphi]\!]_{\mathcal{K}})$ $\quad$ $[\![\overleftarrow{\Box}\varphi]\!]_{\mathcal{K}} := \mathsf{suc}^{\mathcal{R}}_{\forall}([\![\varphi]\!]_{\mathcal{K}})$

$[\![\mu x.\varphi]\!]_{\mathcal{K}} := \bigcap\{Q \subseteq \mathcal{S} \mid [\![\varphi]\!]_{\mathcal{K}^Q_x} \subseteq Q\}$ $\quad$ $[\![\nu x.\varphi]\!]_{\mathcal{K}} := \bigcup\{Q \subseteq \mathcal{S} \mid Q \subseteq [\![\varphi]\!]_{\mathcal{K}^Q_x}\}$

The satisfying states of fixpoint formulae can be computed by fixpoint iteration: The states $[\![\mu x.\varphi]\!]_{\mathcal{K}}$ satisfying a least fixpoint formula $\mu x.\varphi$ are obtained by the iteration $Q_{i+1} := [\![\varphi]\!]_{\mathcal{K}_x^{Q_i}}$ starting with $Q_0 := \emptyset$. For greatest fixpoint formulae, the iteration starts with $Q_0 := \mathcal{S}$. In both cases, the sequence Q_i is monotonic (increasing for least fixpoints and decreasing for greatest ones).

An important characteristic of a μ-calculus formula is its *alternation-depth*, which is roughly speaking the number of interdependent fixpoints. For example, the formula $\mu y.\Box(\nu x.((y \vee \varphi) \wedge \Box x))$ has alternation-depth two, since the inner fixpoint depends on the outer one. A formula that does not contain interdependent fixpoints is *alternation-free*.[2] The importance of the alternation-depth stems from the fact that the complexities of all known model checking algorithms for the μ-calculus are exponential in it [18]. Regarding the above formula, this means that for each iteration of the outer fixpoint formula, the inner one has to be reevaluated.

Let us now return to the problem of stability checking. Suppose that φ_{in} holds iff the inputs of an SAS are stable for one unit of time. Moreover, let φ_{so} hold iff the state variables and the outputs are stable for one time unit. Then, the SAS is stable iff the LTL formula $\Phi :\equiv \mathsf{AG}(\mathsf{G}\varphi_{\mathsf{in}} \rightarrow \mathsf{FG}\varphi_{\mathsf{so}})$ holds. However, simply checking Φ by means of standard model checking procedures for LTL is not optimal, since the resulting μ-calculus formula is not alternation-free. The proof is based on the fact that Φ is equivalent to $\mathsf{A}(\mathsf{FG}\varphi_{\mathsf{in}} \rightarrow \mathsf{FG}\varphi_{\mathsf{so}})$. Given that φ_{in} holds on all states, we obtain the formula $\mathsf{AFG}\varphi_{\mathsf{so}}$. This formula can be translated to the μ-calculus formula $\mu y.\Box(\nu x.((y \vee \varphi_{\mathsf{so}}) \wedge \Box x))$ [13,18], which has has alternation-depth two (see above). In the following, we propose a solution that does not require the computation of interdependent fixpoints and turns out to be more efficient in practice. For that purpose, we need the notion of paths of a Kripke structure:

Definition 9 (Paths). *A path $\pi\colon \mathbb{N} \rightarrow \mathcal{S}$ of a Kripke structure $\mathcal{K} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L})$ is an infinite sequence of states such that $(\pi(i), \pi(i+1)) \in \mathcal{R}$ holds for all $i \in \mathbb{N}$. The set of all paths originating in a state $s \in \mathcal{S}$ is denoted by $\mathsf{Paths}_{\mathcal{K}}(s)$.*

We assume that the set of variables $\mathcal{V}$ consists of a set of input variables $\mathcal{V}_{\mathsf{in}}$ and a set of state and output variables $\mathcal{V}_{\mathsf{so}}$ such that $\mathcal{V}_{\mathsf{in}} \cap \mathcal{V}_{\mathsf{so}} = \emptyset$ holds. Moreover, we say that a path is steady w.r.t. a set of variables iff none of the variables changes its value after some point of time.

Definition 10 (Steadiness). *Given a Kripke structure $\mathcal{K} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L})$, a path $\pi\colon \mathbb{N} \rightarrow \mathcal{S}$ is steady from time $k \in \mathbb{N}$ w.r.t. a set of variables $V \subseteq \mathcal{V}$ iff the following holds:*

$$\mathsf{steady}_{\mathcal{K}}(\pi, k, V) :\Leftrightarrow \forall i \geq k.\mathcal{L}(\pi(i)) \cap V = \mathcal{L}(\pi(i+1)) \cap V$$

[2] As every CTL formula can be translated to an alternation-free μ-calculus formula [6,18], there is no need for specialised verification procedures in order to check the generic properties described in Subsection 3.2.

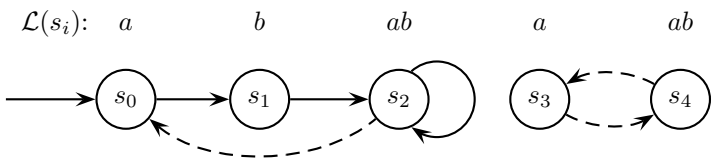

Fig. 3. Example for a Kripke structure

As an example, consider the Kripke structure shown in Figure 3 (dashed lines represent transitions of the restricted transition relation $\mathcal{R}'$ defined below). Since b holds on states s_1 and s_2, the path $s_0, s_1, s_2, s_2, \ldots$ is steady w.r.t. the set $\{b\}$ from time one. In contrast, the path $s_3, s_4, s_3, \ldots$ is not steady w.r.t. $\{b\}$, as b holds on state s_4, but not on state s_3.
Having defined the notion of steadiness, stability of a state and a Kripke structure can now be rephrased as follows:

Definition 11 (Stability). *Given a Kripke structure $\mathcal{K} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L})$, a state $s \in \mathcal{S}$ is stable iff we have:*

$$\mathsf{stable}_{\mathcal{K}}(s) :\Leftrightarrow \forall \pi \in \mathsf{Paths}_{\mathcal{K}}(s).\mathsf{steady}_{\mathcal{K}}(\pi, 0, \mathcal{V}_{\mathsf{in}}) \rightarrow \exists i \in \mathbb{N}.\mathsf{steady}_{\mathcal{K}}(\pi, i, \mathcal{V}_{\mathsf{so}})$$

Moreover, $\mathcal{K}$ is stable iff every reachable state of $\mathcal{K}$ is stable.

Consider again the Kripke structure of Figure 3 and assume that $\mathcal{V}_{\mathsf{in}} = \{a\}$ and $\mathcal{V}_{\mathsf{so}} = \{b\}$ holds. Then, the states s_0, s_1, and s_2 are stable as all paths originating in these states are either not steady w.r.t. $\mathcal{V}_{\mathsf{in}}$ or steady w.r.t. $\mathcal{V}_{\mathsf{so}}$. Since the path $s_3, s_4, s_3, \ldots$ is steady w.r.t. $\mathcal{V}_{\mathsf{in}}$ but not w.r.t. $\mathcal{V}_{\mathsf{so}}$, s_3 and s_4 are not stable.

In order to formulate stability as a μ-calculus formula without interdependent fixpoints, we first restrict the transition relation to those paths that are steady w.r.t. $\mathcal{V}_{\mathsf{in}}$ and do not contain any self-loops, i.e., we construct a Kripke structure $\mathcal{K}' = (\mathcal{S}, \mathcal{I}, \mathcal{R}', \mathcal{L})$ with

$$\mathcal{R}' := \{(s, s') \mid (s, s') \in \mathcal{R} \wedge s \neq s' \wedge \mathcal{L}(s) \cap \mathcal{V}_{\mathsf{in}} = \mathcal{L}(s') \cap \mathcal{V}_{\mathsf{in}}\}.$$

Then, it remains to check whether the paths of $\mathcal{K}'$ are steady w.r.t. $\mathcal{V}_{\mathsf{so}}$. As $\mathcal{R}'$ does not contain any self-loops, a path π is steady from time k iff $\pi(k)$ has no successors. Thus, a state $s \in \mathcal{S}$ is stable iff all paths originating in s are finite. In the μ-calculus, this can be expressed by the formula $\nu x.\Diamond x$, which holds in a state $s \in \mathcal{S}$ iff there exists at least one infinite path originating in s. Consequently, a Kripke structure $\mathcal{K}$ is stable iff $[\![\nu x.\Diamond x]\!]_{\mathcal{K}'}$ does not contain any reachable states. The latter are exactly the set $[\![\mu x.\chi_{\mathcal{I}} \vee \overleftarrow{\Diamond} x]\!]_{\mathcal{K}}$, where $\chi_{\mathcal{I}}$ is the characteristic function of $\mathcal{I}$, i.e., $[\![\chi_{\mathcal{I}}]\!]_{\mathcal{K}} = \mathcal{I}$. This leads to the following theorem:

Theorem 1. *A Kripke structure $\mathcal{K} = (\mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L})$ is stable iff $[\![\mu x.\chi_{\mathcal{I}} \vee \overleftarrow{\Diamond} x]\!]_{\mathcal{K}} \cap [\![\nu x.\Diamond x]\!]_{\mathcal{K}'} = \emptyset$ holds.*

As both fixpoint formulae are independent of each other, they can be evaluated separately, which significantly reduces the total number of fixpoint iterations.

In fact, if a Kripke structure is not stable, we do not even have to compute all reachable states. Given that the formula $\nu x.\Diamond x$ has already been evaluated, the fixpoint iteration for the least fixpoint formula can be aborted when a state is encountered that belongs to $[\![\nu x.\Diamond x]\!]_{\mathcal{K}'}$.

For the Kripke structure of Figure 3, we obtain $[\![\mu x.\chi_{\mathcal{I}} \vee \overleftarrow{\Diamond} x]\!]_{\mathcal{K}} = \{s_0, s_1, s_2\}$ with $\mathcal{I} = \{s_0\}$ and $[\![\nu x.\Diamond x]\!]_{\mathcal{K}'} = \{s_3, s_4\}$. Hence, the Kripke structure is stable.

4 Implementation and Experimental Results

4.1 Modelling Environment and Formal Representation

We integrated the MARS modelling concepts into the Generic Modelling Environment GME[3] [9], a tool for computer-aided software engineering, and developed a GME meta model for representing the MARS modelling concepts. Based on this meta model, concrete examples like our running example (see Figure 4) can be instantiated. GME automatically produces a model representation in XML format which is used as input for validation and verification as well as for code generation.

Besides formal verification of adaptation behaviour, MARS currently offers two further analyses for its validation. First, we support the simulation of adaptation behaviour and the visualisation of reconfiguration sequences using adaptation sequence charts ASC [21]. Second, it is possible to perform a probabilistic analysis of the adaptation behaviour [1]. For this purpose, the adaptation behaviour model is transformed into an equivalent hybrid component fault tree. The probability that a configuration of a module is activated can then be derived from the failure rates of sensors and actuators.

After the adaptation behaviour of a model has been successfully validated and verified, the functional behaviour can be integrated into the model. When the behaviour of the whole system is completely specified, code generation is possible. For simulation and code generation, we use MATLAB-Simulink,[4] the de facto standard in industrial development of embedded systems.

Moreover, we implemented a tool called AMOR (Abstract and MOdular verifieR) that reads XML output generated by GME and translates it to a formal representation based on SAS. SAS models are internally represented as immutable terms using Katja [11]. Additionally, we implemented the transformations described in Subsection 3.1 in order to make the models amenable to formal verification using model checking. The correctness of the transformations is established by automatically generating proof scripts for Isabelle/HOL [12]. AMOR also supports the translation of SAS models into a semantical representation of Isabelle/HOL, so that SAS models can be directly verified using theorem proving techniques. Alternatively, AMOR is able to generate code for symbolic model checkers. The generated code may contain only adaptive behaviour (for the verification of purely adaptive properties) or both adaptive and functional behaviour (for the verification of combined properties).

[3] `http://www.isis.vanderbilt.edu/projects/gme/`

[4] `http://www.mathworks.com`

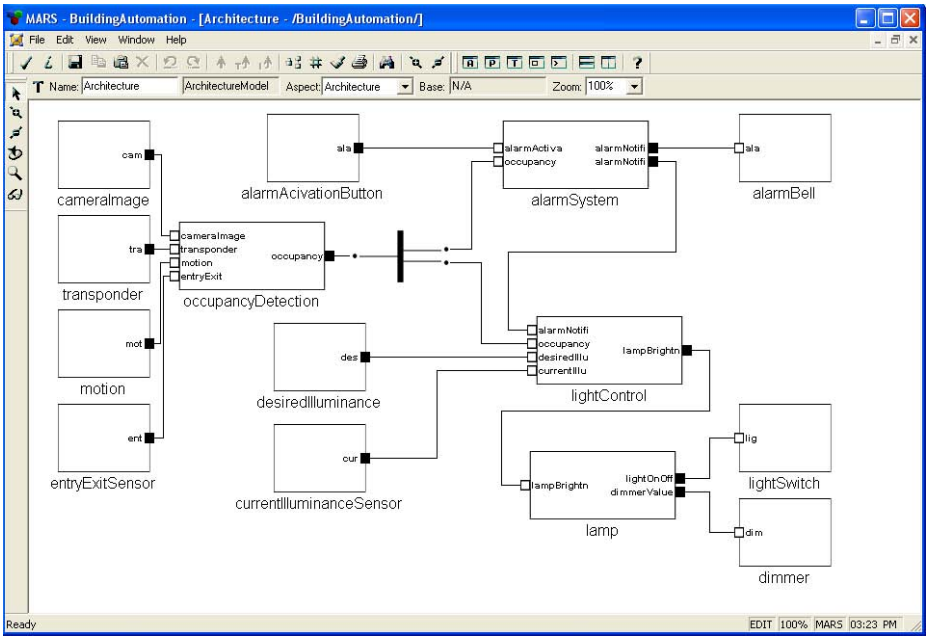

Fig. 4. Top level view of building automation example in GME

4.2 Verification

As model checking back-end, we use the Averest[5] framework, a set of tools for the specification, implementation and verification of reactive systems [19]. In Averest, a system is given in the synchronous programming language Quartz, which is well-suited for describing adaptive systems obtained from SAS models. In particular, as both are based on a synchronous semantics, SAS modules can be easily mapped to threads in Quartz. Moreover, causality analysis of synchronous programs can be used to detect cyclic dependencies that may occur if the quality flow generated by an output is an input of the same module. Specifications can be given in temporal logics as well as in the μ–calculus. To check stability, we implemented the method described in Subsection 3.3 in Averest.

4.3 Evaluation of the Building Automation System

To evaluate our approach, we modelled the building automation example with MARS, translated it to an SAS model using AMOR and generated a Quartz program from this model. The resulting system contains 108 variables and has approximately 1.5×10^{20} reachable states. As the first step, we checked the generic specifications described in Subsection 3.2 for each module using Averest's symbolic model checker. Each of these specifications could be checked in less

[5] `http://www.averest.org`

than one second. For example, we verified the following CTL formulae for the *OccupancyDetection* module:

$$\mathsf{AG}(\texttt{occupancyDetection} = \texttt{off} \rightarrow \mathsf{EF}(\texttt{occupancyDetection} \neq \texttt{off}))$$

$$\begin{aligned}\mathsf{AG}(&\mathsf{EF}(\texttt{occupancyDetection} = \texttt{camera}) \wedge \mathsf{EF}(\texttt{occupancyDetection} = \texttt{transponder}) \wedge \\ &\mathsf{EF}(\texttt{occupancyDetection} = \texttt{motion}) \wedge \mathsf{EF}(\texttt{occupancyDetection} = \texttt{entryExit}) \wedge \\ &\mathsf{EF}(\texttt{occupancyDetection} = \texttt{off}))\end{aligned}$$

$$\begin{aligned}\mathsf{AG}(&\texttt{occupancyDetection} = \texttt{camera} \vee \texttt{occupancyDetection} = \texttt{transponder} \vee \\ &\texttt{occupancyDetection} = \texttt{motion} \vee \texttt{occupancyDetection} = \texttt{entryExit} \vee \\ &\texttt{occupancyDetection} = \texttt{off})\end{aligned}$$

Additionally, we checked five application specific (functional) properties. For instance, the following formula states that if the light is available and the desired brightness is greater than zero then the light will be switched on ($\mathsf{AX}\varphi$ holds iff φ holds on all paths at the next point of time):

$$\mathsf{AG}(\texttt{lightQuality} = \texttt{available} \wedge \texttt{lampBrightness} > 0 \rightarrow \mathsf{AX}(\texttt{light} = \texttt{on}))$$

The application specific properties could also be checked in a few seconds, but the construction of the transition relation required significantly more time compared to the generic properties (92s instead of approx. 1s). This indicates that the separation of adaptive from functional behaviour considerably accelerates verification. As the second step, we checked stability of the system with LTL model checking and with the approach described in Subsection 3.3. LTL model checking requires a total number of 39 fixpoint iterations and takes 130s, whereas our approach only performs 9 iterations in less than one second.

5 Related Work

There are various approaches that integrate model-based design and formal verification in the development of *non*-adaptive systems. Most of them use an intermediate representation that aims at closing the gap between modelling and verification. The Rhapsody UML Verification Environment [17] supports the verification of UML models using the VIS model checker via an intermediate language called SMI. The authors of [22] propose an approach linking xUML, an executable subset of UML, and the SPIN model checker. They also propose transformations on the intermediate layer but do not prove them correct, since the intermediate representation has no formal semantics. The IF Toolset [3] integrates modelling in UML and SDL (Specification and Description Language) with different verification tools using the IF intermediate language. IF also supports a number of techniques to reduce the state space, e.g. elimination of irrelevant parts of a model, but the transformations are not explicitly verified. Furthermore, none of these approaches considers adaptation.

With respect to adaptive system development, most approaches concentrate either on modelling or on verification aspects. There are various approaches focussing on modelling self-managed dynamic software architectures; for a survey,

consult [4]. However, only few of them deal with *predetermined dynamic software reconfiguration* [21] and consider the overall development process of adaptive embedded systems. The method described in [14] addresses the modelling of reconfiguration but omits verification aspects completely. In [24], the authors introduce a method for constructing and verifying adaptation models using Petri nets. However, specifying adaptation behavior using Petri nets is not an intuitive way to design complex industry sized systems like the ESP (Electronic Stability Program). Moreover, the notion of adaptivity is more coarse-grained than in our work, since it is restricted to three fixed types of adaptation.

Regarding verification of adaptive systems, linear time temporal logic is extended in [23] with an 'adapt' operator for specifying requirements on the system before, during and after adaptation. An approach to ensure correctness of component-based adaptation was presented in [8], where theorem proving techniques are used to show that a program is always in a correct state in terms of invariants. Initial work on the verification of MARS models can be found in [20]. In contrast to [20], the work presented in this paper supports the verification of both adaptive and functional aspects. Furthermore, the formal representation introduced in this work bridges the gap between modelling and verification techniques and integrates formal verification into the development process in a way transparent to the user. Additionally, [20] does not discuss specialised verification procedures for properties characteristic for adaptive systems such as stability.

6 Conclusion and Future Work

Although dynamic adaptation significantly complicates system design, it is frequently used as cost-efficient solution to increase dependability in safety-critical embedded systems. In this paper, we have presented an integrated framework for model-based development of adaptive embedded systems that supports intuitive modelling as well as efficient formal verification. It provides the developer with a user-friendly modelling method for specifying the system's adaptation behaviour and its interface to functional behaviour. It further allows to formally represent the semantics of the specified models close to the introduced modelling concepts. This enables us to express crucial system properties in a semantically exact manner. Based on the formal model, these properties can be verified using interactive theorem proving, symbolic model checking and specialised verification methods for adaptive embedded systems.

We are currently extending the modelling concepts for adaptation behaviour by integrating a configuration transition management. Moreover, we are working on additional techniques to reduce verification complexity for the application of model checking such as predicate abstraction. Additionally, we plan to support the development of distributed adaptive systems, where adaptation can be used to compensate for the failure of whole components.

References

1. Adler, R., Förster, M., Trapp, M.: Determining configuration probabilities of safety-critical adaptive systems. In: UbiSafe 2007, IEEE Computer Society Press, Los Alamitos (2007)
2. Blech, J.O., Schaefer, I., Poetzsch-Heffter, A.: Translation validation for system abstractions. In: RV 2007, Vancouver, Canada (2007)
3. Bozga, M., Graf, S., Ober, I., Ober, I., Sifakis, J.: The IF toolset. In: Bernardo, M., Corradini, F. (eds.) Formal Methods for the Design of Real-Time Systems. LNCS, vol. 3185, pp. 237–267. Springer, Heidelberg (2004)
4. Bradbury, J.S., Cordy, J.R., Dingel, J., Wermelinger, M.: A survey of self-management in dynamic software architecture specifications. In: WOSS 2004, pp. 28–33. ACM Press, Newport Beach, USA (2004)
5. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. ACM Transactions on Programming Languages and Systems (TOPLAS) 16(5), 1512–1542 (1994)
6. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT, London, England (1999)
7. Dams, D., Gerth, R., Grumberg, O.: Abstract interpretation of reactive systems. ACM Trans. Program. Lang. Syst. 19(2), 253–291 (1997)
8. Kulkarni, S.S., Biyani, K.N.: Correctness of component-based adaptation. In: Crnković, I., Stafford, J.A., Schmidt, H.W., Wallnau, K. (eds.) CBSE 2004. LNCS, vol. 3054, pp. 48–58. Springer, Heidelberg (2004)
9. Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason, C., Nordstrom, G., Sprinkle, J., Volgyesi, P.: The generic modeling environment. In: WISP 2001, Budapest, Hungary, IEEE Computer Society Press, Los Alamitos (2001)
10. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering 26(1) (2000)
11. Michel, P.: Redesign and enhancement of the Katja system. Technical Report 354/06, University of Kaiserslautern (October 2006)
12. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
13. Niwiński, D.: Fixed points vs. infinite generation. In: LICS 1988, pp. 402–409. IEEE Computer Society Press, Washington, DC. (1988)
14. Rawashdeh, O.A., Lumpp Jr., J.E.: A technique for specifying dynamically reconfigurable embedded systems. In: IEEE Conference Aerospace, IEEE Computer Society Press, Los Alamitos (2005)
15. Schaefer, I., Poetzsch-Heffter, A.: Towards modular verification of stabilisation in self-adaptive embedded systems. In: Datta, A.K., Gradinariu, M. (eds.) SSS 2006. LNCS, vol. 4280, pp. 584–585. Springer, Heidelberg (2006)
16. Schaefer, I., Poetzsch-Heffter, A.: Using abstraction in modular verification of synchronous adaptive systems. In: Workshop on Trustworthy Software, Saarbrücken, Germany (2006)
17. Schinz, I., Toben, T., Mrugalla, Chr., Westphal, B.: The Rhapsody UML Verification Environment. In: SEFM, pp. 174–183 (2004)
18. Schneider, K.: Verification of Reactive Systems – Formal Methods and Algorithms. Texts in Theoretical Computer Science (EATCS Series). Springer, Heidelberg (2003)

19. Schneider, K., Schuele, T.: Averest: Specification, verification, and implementation of reactive systems. In: ACSD 2005, St. Malo, France (2005)
20. Schneider, K., Schuele, T., Trapp, M.: Verifying the adaptation behavior of embedded systems. In: SEAMS 2006, Shanghai, China, pp. 16–22. ACM Press, New York (2006)
21. Trapp, M., Adler, R., Förster, M., Junger, J.: Runtime adaptation in safety-critical automotive systems. In: SE 2007, ACTA, Innsbruck, Austria (2007)
22. Xie, F., Levin, V., Kurshan, R.P., Browne, J.C.: Translating software designs for model checking. In: FASE, pp. 324–338 (2004)
23. Zhang, J., Cheng, B.H.C.: Specifying adaptation semantics. In: WADS 2005, pp. 1–7. ACM, St. Louis, USA (2005)
24. Zhang, J., Cheng, B.H.C.: Model-based development of dynamically adaptive software. In: ICSE 2006, Shanghai, China, pp. 371–380. ACM Press, New York (2006)