Verifying the Adaptation Behavior of Embedded Systems*

Klaus Schneider, Tobias Schuele
Department of Computer Science
University of Kaiserslautern
P.O. Box 3049
67653 Kaiserslautern, Germany
Klaus.Schneider@informatik.uni-kl.de
Tobias.Schuele@informatik.uni-kl.de

Mario Trapp
Fraunhofer Institute for
Experimental Software Engineering (IESE)
Fraunhofer-Platz 1
67663 Kaiserslautern, Germany
mario.trapp@iese.fraunhofer.de

ABSTRACT

Many complex embedded systems dynamically adapt their components, services, algorithms, and parameters to the environment. This leads to new classes of design errors, since adaptation has become an increasingly complex part of the systems' behavior. In particular, as adaptations often continuously trigger further adaptations in other components, inconsistent and unstable configurations may be reached. Formal verification, which is routinely applied in safety–critical applications, must therefore consider not only temporal and functional properties of a system, but also its ability to dynamically adapt itself according to external and internal stimuli.

In this paper, we describe how the adaptation behavior of embedded systems can be modeled, specified, and verified at design time. The systems are thereby given at a high level of abstraction, where adaptation is triggered by the quality of data values. This allows to extract the relevant information in a form that can be directly used for verification. Moreover, we demonstrate how state-of-the-art model checkers can be used to formally reason about the resulting system description.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking*

General Terms

Design, Verification, Reliability

*The work presented in this paper has been partially carried out in the BelAmI project, funded by the German Federal Ministry of Education and Research (BMBF), the Fraunhofer–Gesellschaft, and the Ministry of Science, Education, Research and Culture (MWWFK) of Rheinland–Pfalz. Moreover, it is part of the Research Excellence Cluster 'Dependable Adaptive Systems and Mathematical Modeling' funded by the Ministry of Science, Education, Research and Culture (MWWFK) of Rheinland–Pfalz.

SEAMS'06, May 21–22, 2006, Shanghai, China.

Keywords

Adaptive embedded systems, verification, model checking

1. INTRODUCTION

Embedded systems are used in many devices to achieve various goals like reduction of production costs, addition of intelligent functionalities, optimization of the entire system, and personalization for specific needs. For example, modern cars contain about one hundred embedded systems for different tasks such as antilock braking (ABS), vehicle stability control (ESP), and adaptive cruise control (ACC).

The complexity of embedded systems grows exponentially as the power of microprocessors grows according to Moore's law. However, in many applications it can be observed that, depending on the current situation, only few parts of a system are required at particular points of time. For this reason, the costs of embedded systems can be significantly reduced by *dynamically adapting* the systems according to the currently required needs. For example, embedded systems in buildings have to use motion detectors instead of cameras to check the presence of persons in the building when it is dark.

The adaptation of a component in a system often triggers *chain reactions* causing further adaptations in other components. Complex problems may result from these chain reactions like infinite triggering of new adaptations or inconsistent configurations in different components. A major challenge for analyzing the adaptation behavior of embedded systems is the immense complexity caused by interdependencies between the components. For this reason, *abstraction methods* are required to extract the relevant information. In this paper, we describe a technique to extract such an *abstract model of the adaptation behavior* that lends itself well for *formal verification*. In this way, essential properties like persistence or fairness of the adaptation can be formally verified.

The key to obtain an abstract model of the adaptation behavior is to endow the data flow between the components by *quality descriptions*. The adaptation behavior of a single component is thereby described by configuration rules that only consider the quality of the input and output variables. As the information to trigger an adaptation is locally available, our method supports modular modeling of reusable adaptive components. Considering only the attached quality descriptions and their effects on reconfigurations allows us to extract an abstract model of the adaptation behavior. As an advantage, the abstract models can be directly used to formally reason about different kinds of properties like races, deadlocks, livelocks, stability etc.

Once such a model has been established, our tool Chameleon [25] can be used to generate a system description for our verification framework Averest [22],[20],[21]. For that purpose, it is important that Averest is able to read system descriptions given as synchronous programs [1],[11]. This allows the precise specification of temporal properties and to distinguish between different priorities of events. Moreover, concurrently invoked actions are easily modeled due to the true concurrency supported by synchronous languages.

As the next step, the desired properties of the adaptation behavior can be specified using temporal logics [17] such as CTL and LTL. The specifications can then be checked using Averest that supports various symbolic model checking techniques and even combinations thereof such as bounded, unbounded, global, and local model checking. Finally, Averest can be used to generate C code in order to obtain an executable model for simulating the system.

The rest of this paper is organized as follows: Next, we discuss related work on adaptive systems and formal verification. Then, we describe our running example (Section 3) and the modeling of adaptive embedded systems based on the quality flow concept (Section 4). In Section 5, we present our verification tool and in Section 6 we illustrate the approach by means of a small case study. Finally, we conclude with a summary and directions for future work.

2. RELATED WORK

2.1 Adaptive Systems

Regarding dynamic adaptation from a general point of view, many results have been obtained in various research areas including real time systems [10], agent systems [13], and component middleware [9] to name only some representatives. Explicit *modeling and analysis of dynamic adaptation in embedded systems*, however, is quite a young research area and only a few research groups already focused on this topic. Even in this scope, currently ongoing research is focused on frameworks that mainly aim at *realizing* the adaptation in embedded systems, but neglect its analysis.

In the RoSES project [15],[24], a reconfiguration framework for embedded systems has been developed. However, the RoSES project does not consider the modeling of dynamic adaptation. Instead, its objective concerning dynamic adaptation is to leave all configuration decisions to the system. On the one hand, the developer does not need to care about the reconfiguration process, but on the other hand, the adaptation behavior can hardly be analyzed. Moreover, dynamic adaptation is essentially restricted to the architectural level.

The DepAuDE project[1] is not directly focused on dynamic adaptation. However, the separate specification of functionality and fail–over mechanisms has been taken into account. The project aims to support reusable, separate specifications of well–known fail–over strategies like watchdogs or voters.

In [6], so–called containment units were introduced that permanently monitor the quality of functional units. If a quality does not meet the defined requirements, a containment unit tries to switch the system to an alternative variant in order to improve the current situation. Again, [6] only considers the realization of dynamic adaptation but not the underlying modeling and analysis concepts.

[1] `http://www.depaude.org`

In [8], an agent–based reconfiguration framework was presented that supports dynamic adaptation of parameters, functional variants, and code migration.

Dynamically reconfigurable hardware [12],[7] is more and more used in hardware designs. The main motivation in this area is to use dynamic reconfiguration in order to adapt to application specific needs that may vary during runtime. Examples are reconfigurable instruction sets of processors, reconfigurable functional units of processors, and dynamically adaptive hardware components like cache memories.

The electronic vehicle stability control program ESP of the Robert Bosch GmbH already realizes dynamic reconfiguration of the system [27],[28],[26]. In order to manage the complexity of dynamic adaptation, the modeling concepts illustrated in this paper have been used to systematically derive an abstract model of the adaptation behavior used as input for the reconfiguration framework of the ESP.

2.2 Formal Verification

As mentioned previously, the adaptation behavior of embedded systems can be very involved. Due to the complexity of such systems, simulation and testing alone are no longer sufficient to gain a sufficiently high quality of the designs. Moreover, in safety–critical applications it is mandatory to formally prove that the system meets the given specifications.

The verification of *finite state systems*, e.g. hardware circuits, is one of the success stories of computer science. The breakthrough was achieved in the early nineties, where it was observed that finite sets can be efficiently represented by means of binary decision diagrams (BDDs), a canonical normal form for propositional logic formulas [2]. The development of BDDs was a cornerstone for *symbolic model checking* procedures based on fixpoint computations [3] (see textbooks like [4],[17] for more details). With sophisticated implementations and refinements of symbolic model checking, it has become possible to verify systems of industrial size, and to detect errors that can hardly be found using simulation [5].

However, finding a suitable abstract model for verification is often a nontrivial task. Sophisticated techniques like quotient computations by bisimulation equivalences, symmetry reductions, partial order reductions, and abstract interpretation are used for that purpose [4],[17]. In the following, we show that the abstract models for describing the adaptation behavior of embedded systems given in [25] can be directly used for verification.

To the best of our knowledge, there is no previous work on the verification of adaptation behavior. We aim at closing this gap by modeling the adaptation behavior of embedded systems in a systematic way so that models for verification can be generated automatically.

3. RUNNING EXAMPLE

In this section, we briefly present an example that we use throughout the paper to illustrate the supported concepts of adaptation and the underlying modeling techniques. These have been successfully applied in several case studies, including industrial systems in the area of vehicle dynamics. For the sake of simplicity, however, we restrict ourselves to a simple example that implements a subset of a building automation system. The system consists of an occupancy detection service, a light control service, and several sensors and actuators. Even in this small system several adaptations may take place, an example scenario is illustrated in Figure 1.

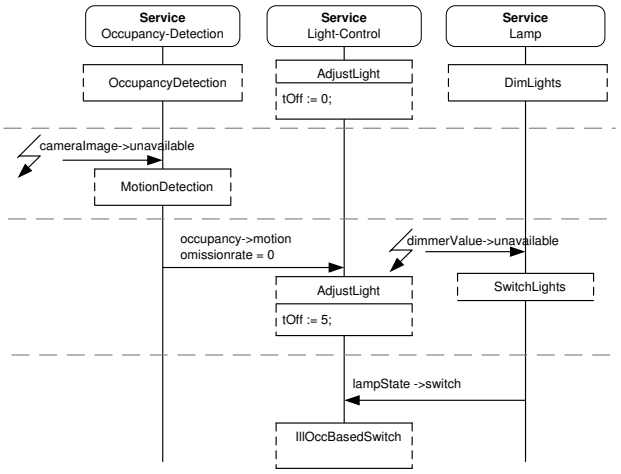

Figure 1: Example for adaptation

Initially, the *occupancy detection* uses a camera image to detect the presence of persons in the monitored room. When the camera fails or is unavailable for other reasons, the occupancy detection uses motion detectors instead. However, this influences the quality of the provided occupancy value since only motions are detected but not the presence of persons. For this reason, a corresponding quality information is propagated to the *light control*. Based on the information that only motions are detected, the light control increases the parameter *tOff* in order to delay the point of time when the lights are switched off in the case that no person is detected. As long as a person is present, it controls the illumination of the room. However, if the dimmer fails, the service which controls the lamps cannot dim but only switch the lights. This information is again propagated to the light control which in turn performs an adaption. In this case, a simple state machine is used that merely switches the lights depending on the current illuminance and the presence of persons.

This simple example illustrates that adaptation of one service may cause a chain reaction of adaptations in other services. In real world applications, it is not unusual that a sensor fault triggers a chain reaction of adaptations that influence more than 80% of the services (cf. [25]). Such chain reactions are too complex to be managed manually. Instead, an explicit model of the adaptation behavior is required that can be used as a basis for different kinds of analyses. In particular, formal methods can be used to verify that the adaptation behavior meets the specification.

4. MODELING ADAPTATION BEHAVIOR

From the viewpoint of the application, we consider a system as a set of concurrently running *services*. In this model, services communicate with each other via shared variables[2] that are tagged with a *quality description*. Services can be reconfigured by regarding the information carried by their input and output variables, i.e., information that is available locally. Based on the current *configuration of a service*, it is possible to define the quality descriptions of its outputs. In this way, quality descriptions are propagated from the event triggering a reconfiguration (e.g. a defect sensor) to all affected services.

[2]The communication may be different in reality. For example, message passing could be used instead shared variables.

4.1 Types, Modes, and Quality Descriptions

The objective of the quality description is to support the developer in the decision how a given quality influences the developed service and how the configuration of the service influences the quality of its output variables. In the course of our research and its practical application [27],[28],[25], it turned out that it is reasonable to *divide the quality description in two levels*:

- First, we define *modes which represent the method that was used to determine the value* of the variable. This includes a description of the characteristics, advantages, drawbacks, and limitations of the used method.
- Second, we define *mode–specific attributes* to quantify a mode–specific degree of quality. By tailoring these attributes to particular modes, very expressive descriptions of quality can be defined.

Consider, for example, the embedded system presented in the previous section that checks the presence/absence of persons in a building. A quality description for a variable `occupancy` can be defined as follows: As the occupancy can be determined by motion detectors or by light barriers, we may use the corresponding modes `motion` and `entryExit`. The former is refined by the attribute `r_point` which defines the precision of the motion detection. The corresponding graphical definition is illustrated in Figure 2.

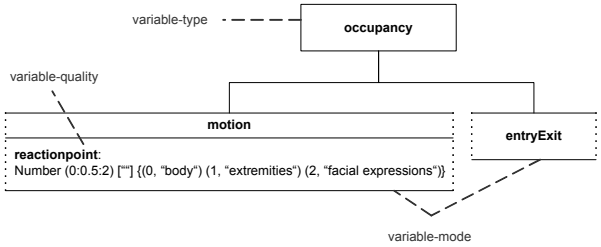

Figure 2: Example for the definition of a variable–type

The developer of a light-control system who uses the variable `occupancy` can now use the provided information as follows: If the variable has the mode `motion`, it may be the case that persons are still present in the building, even if currently no motions are detected. However, it is probable that in this case some motions will be detected after some time. Consequently, it is reasonable to delay switching the lights off for some time unless new motions are detected. Moreover, the value for the delay time can be derived from the reaction point.

4.2 Services

As already explained, services communicate with each other in our model using shared variables that are endowed with quality descriptions. Thus, the information required to define the adaptation behavior of a service is available at its local interface. In order to define adaptive services, some extensions to a common component concept are necessary:

- Ports that are used to send and receive values must be extended in order to additionally support their quality descriptions (cf. Section 4.2.1).
- Services may have different configurations that realize the service (cf. Section 4.2.2). The most appropriate

configuration is chosen at run time depending on the qualities of the required variables.

- In order to achieve a more fine grained adaptation, it is also possible to assign adaptive parameters to each configuration (cf. Section 4.2.3).

4.2.1 Ports

Services are connected via ports that are associated with input and output variables. However, the distinction between input and output ports only allows to consider the data flow between services. The quality flow may use the opposite direction as compared to the corresponding data values. For example, assume a light control service has an output variable `lampBrightness`. The current service configuration may depend on the quality of its output `lampBrightness`, e.g., if it is possible to dim the lights or at least to switch the lights.

For this reason, we have to distinguish between *required and provided variables*. Consequently, each port is additionally marked as required or provided. Regarding the previous example, the light–control services would have a required output for the variable `lampBrightness`.

4.2.2 Configurations

Services may have several configurations, each of which representing a different algorithm to provide the service. For example, the occupancy of a room can be detected using motion detectors, a camera image, a transponder signal, and by registering all entering and leaving persons.

To determine a configuration, a rule is defined for every configuration. Each rule has a *guard*, i.e., a condition that depends on the qualities of the service's variables. Configurations whose guards are satisfied are called *enabled*. As several configurations might be enabled at a time, priorities are assigned to the configurations. If more than one guard is enabled, the configuration with the highest priority is selected.

Configuration	Pr.	Guard
`OccupancyDetection`	4	`cameraImage[available]`
`TransponderDetection`	3	`transponderID[available]`
`MotionDetection`	2	`motion[available(r_point>0)]`
`Off`	1	`true`

Figure 3: Example of configuration rules

Figure 3 shows an example, where the guard for configuration `MotionDetection` has priority 3 and its guard is `motion [available(r_point>0)]`. This means that `motion` is a required variable whose mode must be `available`. Moreover, the attribute `r_point` must be a positive number. If this guard holds, and no guard with higher priority also holds, the corresponding service will adapt to `MotionDetection`.

The guards and priorities therefore determine the configurations of services. Analogously, the quality flow is determined by a list of *influence rules* assigned to each configuration. An influence rule consists of a guard and the actual influence. The first rule the guard of which is true is executed, i.e., is used to determine the mode and the attributes of the provided variables.

As an example, Figure 4 shows an influence rule of the configuration `MotionDetection`. If the variable `motion` is available in a good quality, the motion detection can be considered to be an actual occupancy detection. Consequently, the mode of the provided variable `occupancy` is set to the equally named mode. If the quality of `motion` is not sufficient, the mode of the provided variable `occupancy` is set to `motion` in order to signal that motions instead of occupancies are detected. Additionally, the value of the attribute `r_point` of the required variable `motion` is propagated to the provided variable `occupancy`.

```
motion.r_point>1 => occupancy->occupancy;
occupancy->motion
(r_point := motion.r_point);
```

Figure 4: Example for influence rules

4.2.3 Parameters

In addition to the coarse grained adaptation of services to configurations, it is possible to define adaptive parameters whose values are dynamically determined by the values and quality descriptions of the variables. Parameters are defined in the behavior specification of a configuration, where a function is defined to derive a parameter's value at runtime. Again, this function may use the modes and attributes of the required variables as arguments, which are available at the local service interface.

For example, the configuration `MotionDetection` of an occupancy detection service might have a parameter `delay` that determines the maximal bound on the amount of time before the room is assumed to be unoccupied when no motions are detected during this interval. Obviously, the current value of this parameter depends on the quality of the used variables for motion detection. A possible parameter function is shown in Figure 5.

```
parameter delay {
 delay := round((2-motion.r_point)/2 * 600);
}
```

Figure 5: Example of a dynamically adaptive parameter

4.3 Adaptation Behavior at System Level

Based on configuration and influence rules as well as the parameter functions, the adaptation behavior of single services is completely defined. In order to define an adaptive system that consists of several services, the different services are instantiated and their ports are connected accordingly. If necessary, global segments can be defined that influence the adaptation behavior of a *group of services* in order to optimize the adaptation behavior from a global point of view [25].

5. THE AVEREST SYSTEM

In this section, we describe the Averest framework [22],[20], [21] that provides tools for verifying temporal properties of synchronous programs [1],[11] as well as for compiling these programs to hardware and software systems. In particular, many formal verification techniques, including model checking of temporal properties of finite and infinite state systems are available.

In Averest, a system is described using the Esterel–like synchronous programming language Quartz [16], and specifications can be given in temporal logics such as LTL, CTL, ω-automata, and directly in the μ–calculus [17]. Currently, Averest consists of the following tools that cover large parts of typical design flows:

- Ruby: a compiler for translating Quartz programs to finite and infinite state transition systems
- Beryl: a symbolic model checker for finite and infinite state transition systems
- Topaz: a code generator to convert transition systems into hardware and/or software

Figure 6 shows the typical design flow. A given Quartz program is first translated to a symbolically represented transition system in Averest's Interchange Format AIF that is based on XML. The AIF description can then be used for verification and code generation. Moreover, there are interfaces to third–party tools, e.g. other model checkers such as SMV [14].

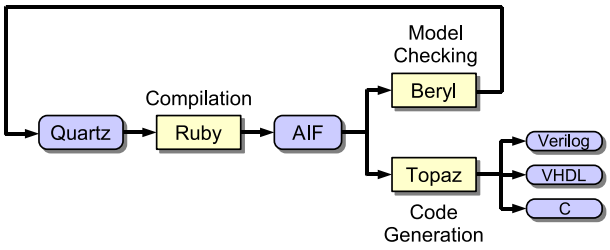

Figure 6: Averest **design flow**

For the work presented in this paper, an additional front–end called Chameleon has been developed. Chameleon [25] can be used to specify the adaptation behavior as outlined in the previous section. Using Chameleon, it is possible to automatically generate synchronous programs as input for Averest in order to formally check certain properties of interest. We consider the translation done by Chameleon in Section 6. In the remainder of this section we describe some background information on Averest focusing on the verification facilities.

5.1 Synchronous Programs

The basic paradigm of synchronous languages [1],[11] is the distinction between *micro and macro steps* in a program. From a programmer's point of view, micro steps do not take time, whereas macro steps take one unit of time. Hence, consumption of time is explicitly programmed by partitioning the program into macro steps. This programming model, referred to as perfect synchrony [1],[11], together with a deterministic form of concurrency allows the compilation of *multi–threaded synchronous programs* to deterministic single–threaded code. In this way, multi–threaded programs can be executed on ordinary microcontrollers without complex operating systems.

A distinct feature of synchronous languages is their *detailed formal semantics* that is usually given by means of transition rules in structural operational semantics. This makes synchronous languages attractive for safety–critical applications where formal verification is mandatory. Moreover, they allow direct translations to synchronous hardware circuits. On the other hand, the synchronous programming model challenges the compilers: Intrinsic problems like *causality and schizophrenia problems* must be solved [18],[19].

In comparison to Esterel, Quartz offers the following additional features:

- generic programs (with compile–time parameters)
- (synchronous, asynchronous, interleaved) concurrency
- explicit nondeterministic choice
- fixed bitwidth (signed and unsigned) integers with a complete set of binary arithmetic operations
- infinite integers (with a restriction to decidable fragments of arithmetic)
- temporal logic specifications given in LTL, CTL, and certain extensions

Besides the translation of Quartz programs to transition systems, Ruby translates temporal logic specifications to alternating ω–automata and/or μ–calculus formulas to simplify the verification process.

5.2 Model Checking

Beryl is a symbolic model checker for the verification of finite and infinite state systems. For the verification of finite state systems, Beryl employs binary decision diagrams [2] as the basic data structure. Analogously, infinite state sets are represented by means of finite state automata that can be used as a canonical normal form for decidable predicate logics.

The ability to deal with data types over infinite domains makes Beryl attractive for the verification of abstract models where implementation specific details such as the bitwidth can be neglected. This is an important step towards the verification in early design phases, as required for the verification of adaptation behavior.

Beryl contains algorithms for global and local model checking, as well as bounded variants thereof [23]. There are many differences between these algorithms that may lead to fundamentally different runtimes. For the verification of infinite state systems, it is even more important to have different model checking algorithms, since termination cannot be guaranteed due to the undecidability of most verification problems [23].

6. CASE STUDY

Even though the properties that have to be considered for specifying the adaptation of an embedded system depend on the application and the system, there are some general questions that often arise:

- Can a certain configuration be reached at all?
- Can every configuration be left or can the system be caught in such a configuration?
- Can a certain configuration be reached infinitely often, i.e., is the system fair?
- Are there dependencies between configurations of different components?
- How long will it take to complete an adaptation? Will it terminate at all?

Properties like the ones above can be easily specified by means of temporal logics [17] so that efficient model checking techniques can be used for their verification. Moreover, in case a property does not hold, modern model checkers are able to generate a trace to demonstrate an adaptation sequence that violates the property. In order to verify a property using Averest, it is necessary to translate the given model of the adaptation behavior to a Quartz program. This transformation step is outlined in the following.

Services are translated to single Quartz modules that run concurrently in a main module as parallel threads. As an example, Figure 7 shows the module for the occupancy detection service as described in Figure 3. Since we analyze only

```
module OccupancyDetection:
input
  cameraImage: boolean,
  transponderID: boolean,
  motion: boolean,
  motion_r_point: integer,
  oc_OccupancyDetection: boolean,
  oc_TransponderDetection: boolean,
  oc_MotionDetection: boolean;
output
  occupancy: integer,
  ocd_config: integer,
  ocd_r_point: integer;
begin
  ocd_config := OCD_Off;
  OccupancyDetectionInit: pause;
  loop
    case
      // ---------------------------------------
      // -- configuration OccupancyDetection
      // ---------------------------------------
      (cameraImage & oc_OccupancyDetection)
      do
        next(occupancy) := OCD_ocd;
        ocd_config := OCD_Occupancydetection;
      // ---------------------------------------
      // -- configuration TransponderDetection
      // ---------------------------------------
      (transponder & oc_TransponderDetection)
      do
        next(occupancy) := OCD_transponderdetection;
        ocd_config := OCD_TransponderDetection;
      // ---------------------------------------
      // -- configuration MotionDetection
      // ---------------------------------------
      (motion & oc_MotionDetection)
      do
        if (motion_r_point > 0) then
          next(occupancy) := OCD_ocd
        else
          next(occupancy) := OCD_motiondetection;
          next(oc_r_point) := motion_r_point
        end;
        ocd_config := OCD_MotionDetection;
      // ---------------------------------------
      // -- configuration Off
      // ---------------------------------------
      else
        next(occupancy) := OCD_unavailable;
        ocd_config := OCD_Off
    end;
    pause
  end
end
```

Figure 7: Synchronous program for occupancy detection

the adaptation instead of the entire functional behavior, the inputs of a Quartz module do not have the types as in the real implementation. Instead, they only hold the quality description of the variable. Therefore, one input is defined for each *required variable* representing its mode like `motion`, and another input is defined for each quality attribute of the required variables, as for example, `motion_r_point` in Figure 7. In the same way, "provided variables" of the services are defined as outputs like `occupancy`.

Additionally, an output variable is defined that represents the current configuration of the service like `ocd_config`. The configuration rules are specified using a case statement, where each case corresponds to a single rule. The conditions of a case statement are the guards of a configuration rule, and the priorities of the rules are preserved by the order of the case distinctions. The influence rules of the corresponding configuration as well as the parameter functions are defined in the body of a case statement as shown for `MotionDetection`.

As can be seen, using Averest as a verification back end for Chameleon is not by accident. In contrast to other model checkers like SMV [14], Averest offers the following important advantages for verifying the adaptation behavior of embedded systems:

- Services can be easily implemented as threads in Quartz.
- In contrast to simple action languages used by most model checkers, Quartz programs allow one to implement difficult dependencies between control flows. In particular, the priorities of the guards used to determine the next configuration of an adaptation can be easily implemented by case statements in Quartz[3].
- Quartz has a precise notion of concurrency and time. Since Quartz programs are deterministic, counterexamples generated by the model checker can be used for simulation to search bugs.
- Synchronous languages distinguish between input and output variables, and therefore between different directions of information flow. As the quality flow generated by an output may become an input of the same service, *causality problems* could be obtained due to cyclic dependencies. Compilation of synchronous languages includes an analysis of such causality problems [18].
- Beryl can handle infinite state systems as well as finite ones, which is not possible with other model checkers. In particular, the attributes used to define the quality flows can thus be modeled at an abstract level.

7. SUMMARY AND CONCLUSION

Dynamic adaptation is frequently used in embedded systems to react to fundamental changes in the environment such as failure of sensors and actuators. Dynamic adaptation not only helps to reduce costs, which is a crucial concern in the design of embedded systems, but also increases dependability, e.g. by switching to special configurations (graceful degradation). However, the adaptation behavior of embedded systems significantly complicates their design and poses several challenges. In particular, the adaptation of a single component can cause a chain reaction of adaptations in other components.

Our approach to cope with these problems is twofold. First, we propose a method that allows to describe the adaptation behavior at an abstract level. For that purpose, the data flow is augmented with quality descriptions which are used by configuration rules to determine potential adaptations. Second, we use the resulting system description as the basis for generating programs in synchronous languages, which are well–suited for modeling concurrent services occurring in the abstract system description). Moreover, they can finally be translated to transition systems that serve as input for symbolic model checkers.

To evaluate our approach, we implemented it in our tools Chameleon and Averest. As a first benchmark, we modeled the system sketched in this paper. As mentioned previously, it consists of an alarm system, a light control, and a unit for determining occupancy. Each service can operate in up to five configurations. The specifications (total number of 23) could be checked in less than two seconds.

[3]Model checkers like SMV also provide case statements, but with a different semantics: Among the enabled guards, one action is chosen nondeterministically.

8. REFERENCES

[1] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[2] R. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[3] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Symposium on Logic in Computer Science (LICS)*, pages 1–33, Washington, D.C., June 1990. IEEE Computer Society.

[4] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT, London, England, 1999.

[5] E. Clarke and J. Wing. Formal methods: State of the art and future directions. Technical Report CMU-CS-96-178, Carnegie Mellon University, September 1996. ftp://reports.adm.cs.cmu.edu/usr/anon/1996/CMU-CS-96-178.ps.

[6] J. Cobleigh, L. Osterweil, A. Wise, and B. Lerner. Containment units: A hierarchically composable architecture for adaptive systems. *ACM SIGSOFT Software Engineering Notes*, 27(6):159–165, November 2002.

[7] K. Compton and S. Hauck. Reconfigurable computing: A survey of systems and software. *ACM Computing Surveys*, 34(2):171–210, June 2002.

[8] K. Dixon, T. Pham, and P. Khosla. Port-based adaptable agent architecture. In P. Robertson, H. Shrobe, and R. Laddaga, editors, *International Workshop on Self-Adaptive Software (IWSAS)*, volume 1936 of *LNCS*, pages 134–142, Oxford, UK, 2000. Springer.

[9] W. Gilani, N. Naqvi, and O. Spinczyk. On adaptable middleware product lines. In *Proceedings of 3rd Workshop an Adaptive and Reflective Middleware*, pages 207–213, Toronto, Ontario, Canada, 2004. ACM Press.

[10] O. Gonzalez, H. Shrikumar, J. Stankovic, and K. Ramamritham. Adaptive fault-tolerance and graceful degradation under dynamic hard real-time scheduling. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 79–89, San Francisco, CA, USA, 1997. IEEE Computer Society.

[11] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer, 1993.

[12] S. Hauck. The roles of FPGA in reprogrammable systems. *Proceedings of the IEEE*, 86(4):615–638, April 1998.

[13] O. Marin, M. Bertier, and P. Sens. DARX - a framework for the fault tolerant support of agent software. In *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 406–418, Denver, Colorado, USA, 2003. IEEE Computer Society.

[14] K. McMillan. The SMV system, symbolic model checking - an approach. Technical Report CMU-CS-92-131, Carnegie Mellon University, 1992.

[15] W. Nace and P. Koopman. Product family approach to graceful degradation. In B. Kleinjohann, editor, *Workshop on Distributed and Parallel Embedded Systems (DIPES)*, pages 131–140, Paderborn, Germany, 2000. Kluwer.

[16] K. Schneider. Embedding imperative synchronous languages in interactive theorem provers. In *Conference on Application of Concurrency to System Design (ACSD)*, pages 143–156, Newcastle upon Tyne, UK, June 2001. IEEE Computer Society.

[17] K. Schneider. *Verification of Reactive Systems – Formal Methods and Algorithms*. Texts in Theoretical Computer Science (EATCS Series). Springer, 2003.

[18] K. Schneider, J. Brandt, and T. Schuele. Causality analysis of synchronous programs with delayed actions. In *Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 179–189, Washington D.C., USA, September 2004. ACM.

[19] K. Schneider, J. Brandt, and T. Schuele. A verified compiler for synchronous programs with local declarations. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 2006.

[20] K. Schneider and T. Schuele. Averest: Specification, verification, and implementation of reactive systems. In *Conference on Application of Concurrency to System Design (ACSD)*, St. Malo, France, 2005. participant's proceedings.

[21] K. Schneider and T. Schuele. A framework for verifying and implementing embedded systems. In *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, Dresden, Germany, 2006. GI/ITG/GMM.

[22] K. Schneider and T. Schuele. The Averest Framework, 2006. `http://www.averest.org`.

[23] T. Schuele and K. Schneider. Bounded model checking for infinite state systems. *Formal Methods in System Design*, 2006.

[24] C. Shelton, P. Koopman, and W. Nace. A framework for scalable analysis and design of system-wide graceful degradation in distributed embedded systems. In *Workshop on Object-Oriented Real-Time Dependable Systems (WORDS)*, pages 156–163. IEEE Computer Society, 2003.

[25] M. Trapp. *Modeling the Adaptation Behavior of Adaptive Embedded Systems*. PhD thesis, University of Kaiserslautern, 2005.

[26] M. Trapp and B. Schürmann. On the modeling of adaptive systems. In *International Workshop on Dependable Embedded Systems*, Florence, Italy, 2003.

[27] M. Trapp, B. Schürmann, and T. Tetteroo. Service-based development of dynamically reconfiguring embedded systems. In M. Hamza, editor, *IASTED International Multi-Conference on Applied Informatics*, pages 935–941, Innsbruck, Austria, 2003. IASTED/ACTA Press.

[28] M. Trapp, B. Schürmann, and T. Tetteroo. Variable-based environment model for gracefully degrading embedded systems. In M. Hamza, editor, *Software Engineering and Applications (SEA)*, Marina Del Rey, USA, 2003. IASTED/ACTA Press.