Using Abstraction in Modular Verification of Synchronous Adaptive Systems

Ina Schaefer and Arnd Poetzsch-Heffter

{inschaef|poetzsch}@informatik.uni-kl.de
Software Technology Group
Technische Universtiät Kaiserslautern
Germany

Abstract. Self-adaptive embedded systems autonomously adapt to changing environment conditions to improve their functionality and to increase their dependability by downgrading functionality in case of failures. However, adaptation behaviour of embedded systems significantly complicates system design and poses new challenges for guaranteeing system correctness, in particular vital in the automotive domain. Formal verification as applied in safety-critical applications must therefore be able to address not only temporal and functional properties, but also dynamic adaptation according to external and internal stimuli.
In this paper, we introduce a formal semantic-based framework to model, specify and verify the functional and the adaptation behaviour of synchronous adaptive systems. The modelling separates functional and adaptive behaviour to reduce the design complexity and to enable modular reasoning about both aspects independently as well as in combination. By an example, we show how to use this framework in order to verify properties of synchronous adaptive systems. Modular reasoning in combination with abstraction mechanisms makes automatic model checking efficiently applicable.

1 Introduction

In the automotive sector, self-adaptive embedded systems are used for instance as antilock braking (ABS), vehicle stability control (VSC), and adaptive cruise control (ACC) systems. They autonomously adapt to changing environment conditions in order to meet high quality requirements, e.g. to offer the best possible service in any kind of driving condition. Furthermore, adaptation increases dependability and fault-tolerance of systems by autonomously up- and downgrading the functionality according to the available resources. This can for instance be changing qualities of environment sensors. However, adaptation in embedded systems significantly complicates system design and poses new challenges for guaranteeing system correctness, in particular vital in the automotive domain. Therefore, formal verification as applied in safety-critical applications must be able to address not only temporal and functional properties, but also dynamic adaptation according to external and internal stimuli.

INTERREG IIIC/e-Bird
Workshop "Trustworthy Software" 2006
http://drops.dagstuhl.de/opus/volltexte/2006/699

In this paper, we introduce a formal semantic-based framework to model, specify and verify functional and adaptation behaviour of self-adaptive embedded systems. The modelling framework is based on state-transition systems and describes adaptation of module behaviour in terms of an adaptation aspect on top of a set of predetermined module configurations. Restricting adaptation to predetermined reconfiguration makes systems predictable and improves analyses results. Our models are synchronous systems as those can capture simultaneously invoked actions by true concurrency. Most approaches formalizing self-adaptation [1] so far focus on structural and architectural adaptation such as adding and removing components instead of behavioural adaptation of single system modules. Furthermore, they intertwine functionality and adaptation. In contrast, the proposed modelling framework decouples functional and adaptive behaviour and provides a clear formal account of both aspects in separation. This reduces the design complexity and enables explicit and uniform reasoning about purely functional, purely adaptive as well as combined properties. We develop a high level modelling framework in which the special features of dynamic reconfiguration, i.e. the behavioural adaptation and the separation of adaptation and functionality, can be observed and reasoned about directly. If these special properties of the considered class of systems would be encoded into another formalism, this high level specific properties are typically lost and cannot be exploited for tailored analyses.

On top of the formal model, we define a specification logic that allows to express functional, adaptive and temporal properties of the system. Since we can describe the behaviour of our systems by a set of execution traces we will adopt a variant of first-order LTL [3] for our purposes. The proposed framework enables modular reasoning through modular specification of systems. A global system property can be decomposed into local properties of single modules entailing the global property. Furthermore, the model allows to incorporate abstraction mechanisms, for instance to reduce unbounded data domains to finite discrete domains. Modularity combined with appropriate abstraction mechanisms facilitates the efficient integration of existing automatic verification techniques such as model checking into the verification process of synchronous adaptive systems. Thus, the verification effort can be reduced by discharging sub-proof goals automatically.

In this paper, we present the application of our modelling, specification and verification framework at an example system confronted with changing qualities of sensor values. This scenario is quite common in the context of embedded automotive systems. Due to restricted hardware resources, the system has to deal with changing sensor qualities by adapting its functionality to the available resources instead of halting the system. Redundant hardware is not applicable due to the inherent limitations in embedded systems. We show how to model such a sensor quality adaptation as synchronous adaptive system. Afterwards, we verify the safety property that despite problems with the sensors the quality of the system output is below a threshold only for a restricted period of time. This exemplary verification shows the use of modular reasoning and abstraction techniques and gives an intuition which mechanisms are necessary for efficient automatic verification of synchronous adaptive systems.

The paper is structured as follows: Section 2 gives a short overview of related work on formal analysis of self-adaptive systems. In Section 3, we will introduce our formal semantic-based model of synchronous adaptive systems illustrated with the running example. In Section 4, we introduce an LTL based logic for specifying properties over these models. In Section 5, we show how to use abstraction techniques and modular verification in order to proof the safety property over the running example, before we conclude the paper in Section 6 with an outlook to future work.

2 Related Work

From a general point of view dynamic adaptation is a very diverse area of research including real time systems [5], agent systems [8] and component middleware [4], just to name a few representatives. There are a number of approaches for modelling self-managed dynamic software architectures in a more or less formal manner, e.g. using graphs, logic or process algebra; for a survey, consult [1]. However, most of these approaches consider mere modelling of systems instead of their verification. Additionally, the focus lies mainly on architectural adaptation instead of behavioural adaptation as considered here. Moreover, adaptive and functional behaviour are often intertwined which does not allow separated reasoning about both aspects.

In [10], models of adaptive synchronous systems separate adaptation from functionality by endowing data flow with qualities. The configuration behaviour of one module only depends on the quality transmitted with the input and output variables. Considering the qualities, an abstract model of the adaptation behaviour is extracted from the system which is analysed via model checking. However, functional behaviour is completely discarded whereas our approach allows to reason about adaptive, functional and combined behaviour as in systems where adaptation depends on functional data. In [12], the authors use a model driven approach to modularly define adaptive systems coming close to the modularity considered here. Starting from a global model and global requirements of the overall system, single domains of adaptation are identified which are designed to satisfy local requirements entailing the global ones. However, the notion of adaptivity is more coarse-grained than in synchronous adaptive systems due to three fixed types of adaptation.

With respect to verification of adaptive systems, in [11] a linear temporal logic is extended with an 'adapt' operator for specifying requirements on the system before, during and after the adaptation. In [6], the authors use an approach based on a transitional-invariant lattice. Using theorem proving techniques they show that before, during and after the adaptation the program is always in a correct state in terms of satisfying the transitional-invariants. However, both approaches use a more coarse-grained notion of adaptation than predetermined behavioural reconfiguration as considered here.

3 Formal Models of Synchronous Adaptive Systems

Synchronous adaptive systems are composed from a set of modules where each module has a set of predetermined behavioural configurations it can adapt to. The selected configuration depends on the status of the module's environment. It is determined by an adaptation aspect defined on top of the functional behaviour. The modules are connected via links between input and output variables. Data and adaptation flow are decoupled and do not follow the same links. Adaptations in one module may trigger adaptations in other modules by internal adaptation signals via the adaptation links. That may lead to a chain reaction of adaptations through the system. The systems are assumed to be open systems with non-deterministic input provided by an environment. Furthermore, they are modelled synchronously as their simultaneously invoked actions are executed in true concurrency.

3.1 Running Example

Before we start with the formal definitions, we will illustrate the general behaviour of synchronous adaptive systems at an example system dynamically reconfiguring dependant on the quality provided by its input sensors. Figure 1 shows an overview of the system structure.

The system consists of two modules. They receive input from three sensors and control one actuator. The sensors may produce results with varying quality due to changing environment conditions. Hence, the sensor input is associated with a confidence level. This confidence level is an integer value which reflects the sensor's input quality. The higher the confidence level is the higher is the reliability of the value. In our example, a confidence level below 50 models low confidence, between 50 and 100 medium confidence and above 100 high confidence. The confidence level can be determined by enhancing the mere sensor with a functional module. This module for instance records the sensor values over some period of time and monitors its changes. If the sensor value changes by a great amount over a short period of time confidence in this sensor is reduced. Another possibility to calculate the confidence level may be to monitor other system parameters. By performing a plausibility check the sensor module can infer the confidence of the input.

The first two sensor inputs are fed into the first system module which selects one of the sensor inputs according to their confidences. In the considered scenario, sensor 1 produces very good results reflecting the value to be measured very closely. But sensor 1 is also very likely to produce very bad results because of environment changes. This is reflected in the attached confidence level. If the confidence falls below 50, the value is no longer guaranteed to be good enough. Then, the second sensor becomes important. It measures the same input source as the first sensor in general providing lower confidence. Hence, the first sensor is mostly preferred over the second. However, the second sensor is more robust which is reflected by the assumption that the confidence never falls below 50. Thus, if the first sensor produces data with low confidence over some period of

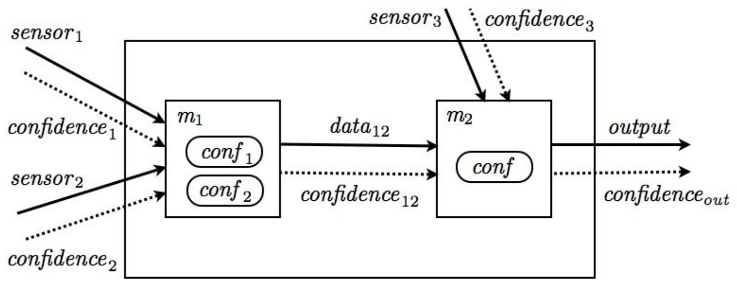

Fig. 1. Graphical Representation of the Running Example System

time the system adapts to use the second sensor in order to ensure sufficient confidence of the output. In detail, the adaptation works as follows: If the confidence level of the first sensor is smaller than 50 for more than 2 subsequent cycles the module switches to the value of the second sensor. Sensor 2 is then used as long as the confidence of the first sensor is smaller than 100. If the confidence of sensor 1 is above 100 for 3 subsequent cycles it is assumed that sensor 1 has recovered. Then, the system will return to using sensor 1 in order to use better quality inputs in general.

According to the selected sensor a different functionality is used to produce the module output. This can for instance be necessary in order to transform the input from a different unit of measurement. The output together with the confidence level of the selected sensor is passed on to the second module which receives another data value and a respective confidence from a third sensor. This sensor is assumed to be of the same type as sensor 2 always producing a medium quality value with confidence above 50. The second module uses its two input values to trigger the actuator. Therefore, it only needs a single configuration. For the confidence level it simply computes the minimum confidence of the received.

An interesting property of this system is that the confidence should never fall below 50 for more than two subsequent cycles. This property depends on the assumption that the second and third sensor are more robust always providing confidence above 50. Ensuring this property is important because the actuator may break down putting the system in a dangerous situation if it gets input with low confidence for more than 2 subsequent cycles. However, it is desirable to use the best possible sensor input. So the adaptation is designed to use sensor 1 whenever appropriate.

3.2 Syntax

In this section, we define the syntax of our formal modelling language for synchronous adaptive systems (SAS). It is based on state-transition systems and incorporates ideas from aspect-oriented software engineering in order to decouple functional from adaptive behaviour. We assume that we are given a set of variable names *Var* and a set of values *Val*. It would also be possible to enhance this with variable types and associated variable domains. The smallest construction element is a module. It contains a set of different predetermined configurations the module can adapt to dependent on the current status of its environment. The adaptation is realised by an adaptation aspect. Before the execution of the actual functionality the adaptation aspect evaluates the configuration guards and determines the configuration to use.

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*
- *$confs = \{conf_j = (guard_j, next\_state_j, next\_out_j) \mid j = 1, ..., n\}$ the configurations of the module, where*
  - *$guard_j$: a first-order formula over $\{in, loc, adapt\_in, adapt\_loc\}$ determining when the configuration j is applicable*
  - *$next\_state_j$: $(in \cup loc \rightarrow Val) \rightarrow (loc \rightarrow Val)$ 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 is defined as a tuple $adaptation = (adapt\_in, adapt\_out, adapt\_loc, adapt\_init, adapt\_next\_state, adapt\_next\_out, adapt\_trigger)$ where

- *$adapt\_in \subseteq Var$, the set of adaptation in-parameters, $adapt\_out \subseteq Var$, the set of adaptation out-parameters, $adapt\_loc \subseteq Var$, 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*
- *$adapt\_next\_out_i : (adapt\_in \cup adapt\_loc \rightarrow Val) \rightarrow (adapt\_out \rightarrow Val)$ the adaptation output function*
- *$adapt\_trigger : (in \cup loc \cup adapt\_in \cup adapt\_loc \rightarrow Val) \rightarrow \{1, ..., n\}$ for n the number of configurations*

Because the first module in our running example has the more interesting adaptation behaviour we will focus on this module for illustrating the modelling framework. Module m_1 possesses two functional inputs $sensor_1$ and $sensor_2$ and the functional output $data_{12}$. Furthermore, it receives the confidence levels from sensor 1 $confidence_1$ and from sensor 2 $confidence_2$ as adaptation inputs and produces $confidence_{12}$ as adaptation output propagating the confidence of the selected sensor. A functional local state does not exist because the module solely transforms input to output according to two configurations, namely configuration $conf_1$ standing for the use of sensor 1 and $conf_2$ for the use of sensor 2.

In Figure 2, the adaptation behaviour, as defined by the $adapt\_next\_state_1$ function, is depicted as a state transition diagram. The adaptation local state

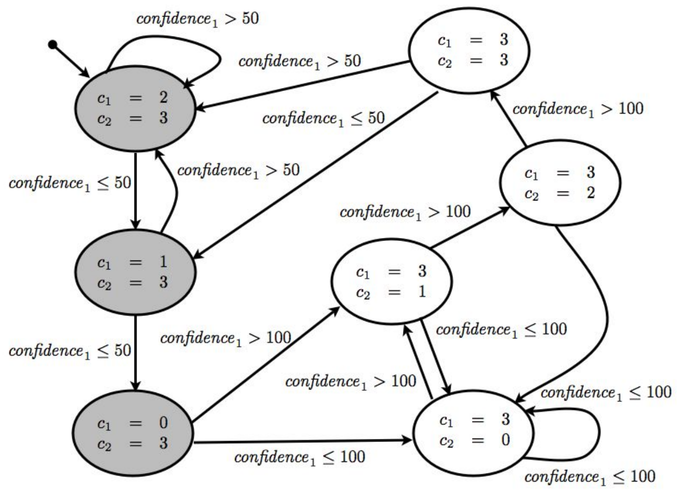

Fig. 2. State Transition Diagram for Adaptation Behaviour of Module 1

consists of two counters c_1 and c_2. If sensor 1 is used counter c_1 counts the subsequent cycles in which $confidence_1$ falls below 50. This counter is initialised to 2. Counter c_2 counts the cycles in which $confidence_1$ is above 100 if sensor 2 is used. It is initialised to 3. The counter c_1 is set to 3 in order to reflect the use of sensor 2. Thus, the guard for use of sensor 1 in $conf_1$ is $c_1 \leq 2$ and the guard for sensor 2 in $conf_2$ is $c_1 = 3$. If sensor 1 is used $confidence_{12} := confidence_1$ and if sensor 2 is used $confidence_{12} := confidence_1$. In Figure 2, the grey circles denote states in which sensor 1 is used and the white ones states where sensor 2 is used.

An SAS system is composed from a set of modules that are interconnected with their input and output variables. The system is an open system with an environment providing non-deterministic input and output via connections from environment input and output to module input and output variables. For technical reasons, we have to assume that the variable names of all modules in a composed system are pairwise disjoint. This can be easily achieved by indexing the module variables with the respective module index. By an injective connection function, we link module output variables to other module's input variables. Furthermore, we link environment input variables to module input variables and module output variables to environment output variables. This means that one variable is connected to one other variable only. If we want to transfer the same output to several places we have to simulate this by duplicating the output variable. Note that in this definition adaptation and functional input and output are decoupled. Adaptation and data flow do not follow the same links such that a module can forward its data to one module and notify a different module to adapt.

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

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

where

- M *is a set of modules* $M = \{m_1, \ldots, m_n\}$ *where* $m_i = (in_i, out_i, loc_i, init_i, confs_i, adaptation_i)$
- $input_a \subseteq Var$ *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 an injective function connecting adaptation outputs to adaptation inputs and also environment adaptation inputs to module adaptation inputs and module adaptation outputs to environment adaptation outputs, i.e.* $conn_a : (adapt\_out_j \rightarrow adapt\_in_k) \cup (input_a \rightarrow adapt\_in_k) \cup (adapt\_out_k \rightarrow output_a)$ *for* $j, k = 1, ..., n$
- $conn_d$ *is an injective function connecting outputs of modules to inputs and also environment inputs to module inputs and module outputs to environment outputs, i.e.* $conn_d : (out_j \rightarrow in_k) \cup (input_a \rightarrow in_k) \cup (out_k \rightarrow output_a)$ *for* $j, k = 1, ..., n$

We can model the running example as

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

where $M = \{m_1, m_2\}$. The adaptation inputs are $input_a = \{confidence_1, confidence_2\}$ and the functional inputs are $input_d = \{sensor_1, sensor_2\}$. The adaptation outputs are $output_a = \{confidence_{out}\}$ and the functional outputs are $output_d = \{output\}$. The connections between the modules are as depicted in Figure 1.

3.3 Semantics

The semantics of an SAS is defined in a two layered approach. Firstly, we define the local semantics of single modules similar to standard state-transition systems. From this, we secondly give the global semantics of the composed system.

A local state of a module is defined by the evaluation of the module's variables, i.e. the input, output and local variables and the adaptation counterparts. A local state is initial, if its functional and adaptation variables are set to their initial values and input and output are undefined. A local transition between two local states evolves in two stages: Firstly, the adaptation aspect computes its new local state and its new adaptation output from the current adaptation input and the previous adaptation state. The adaptation aspect further selects the configuration with the smallest index and valid guard with respect to the current input and the previous functional and adaptation state. Since the configurations are prioritised according to their index we do not require them to be disjoint. The system designer should ensure that the system has a build-in default configuration 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 defined as evaluation of the module's variables.*

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

A local state s is called initial if $s|_{loc} = init$, $s|_{adapt\_loc} = adapt\_init$ and $s|_V = undef$ for all $V = in \cup out \cup adapt\_in \cup adapt\_out$. A local transition between two local states s and s' is defined as $s \to_{loc} s'$ iff

$$s'|_{adapt\_loc} = adapt\_next\_state(s'|_{adapt\_in} \cup s|_{adapt\_loc})$$
$$s'|_{adapt\_out} = adapt\_next\_out(s'|_{adapt\_in} \cup s|_{adapt\_loc})$$
$$s'|_{loc} = next\_state_i(s'|_{in} \cup s|_{loc}) \text{ and } s'|_{out} = next\_out_i(s'|_{in} \cup s|_{loc})$$

and $\quad adapt\_trigger(s'|_{in \cup adapt\_in} \cup s|_{loc \cup adapt\_loc}) = i$

$$\text{iff } s'|_{in} \cup s|_{loc} \cup s|_{out} \cup s'|_{adapt\_in} \cup s|_{adapt\_out} \cup s|_{adapt\_loc} \models guard_i$$
$$\forall\ 0 < j < i, s'|_{in} \cup s|_{loc} \cup s|_{out} \cup s'|_{adapt\_in} \cup s|_{adapt\_out} \cup s|_{adapt\_loc} \not\models guard_j$$

A global system state is the union of the local states of the system modules together with an evaluation of the system's environment input and output. A global system state is initial if all local states are initial and the system input und output are undefined. A transition between two global states is performed in three stages. Firstly, each module reads its input either from another module's output of the previous cycle or from the environment in the current cycle. Secondly, each module synchronously performs a local transition. Thirdly, the modules directly connected to the system output write their results to the output variables.

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

- *for all $x, y \in Var \setminus (input_d \cup input_a)$ with $conn_d(x,y)$ or $conn_a(x,y)$: $\sigma^{i+1}(y) := \sigma^i(x)$ and for all $x \in input_a$ and $y \in Var$ with $conn_a(x,y)$: $\sigma^{i+1}(y) := \sigma^{i+1}(x)$ and for all $x \in input_d$ and $y \in Var$ with $conn_d(x,y)$: $\sigma^{i+1}(y) := \sigma^{i+1}(x)$*
- *for all $s_j^i \in \sigma^i$ and for all $s_j^{i+1} \in \sigma_{i+1}$ $s_j^i \to_{loc} s_j^{i+1}$*
- *for all $x \in Var$ and $y \in output_d$ with $conn_d(x,y)$: $\sigma^{i+1}(y) := \sigma^{i+1}(x)$ and for all $x \in Var$ and $y \in output_a$ with $conn_a(x,y)$: $\sigma^{i+1}(y) := \sigma^{i+1}(x)$*

A sequence of global states $\sigma^0\sigma^1\sigma^2\ldots$ of an SAS is a system trace if firstly σ^0 is an initial global state and secondly, for all $i \geq 0 : \sigma^i \to_{glob} \sigma^{i+1}$. The set $Runs(SAS) = \{\sigma^0\sigma^1\sigma^2\ldots \mid \sigma^0\sigma^1\sigma^2\ldots \mathit{is\ a\ system\ trace}\}$ gives the semantics of the SAS.

4 A Logic for Synchronous Adaptive Systems

In this section, we will introduce a specially tailored logic for reasoning about synchronous adaptive systems. The properties of the system behaviour can be classified in three dimensions: functional behaviour, adaptation behaviour and combined properties. Combined properties equally refer to functional and adaptive system aspects, for instance if adaptation depends on functional values. The environment input is assumed to be non-deterministic such that the behaviour of a system can be described by a set of possible execution traces as infinite sequences of states. Hence, we adopt a variant of the linear time logic LTL [3] by adding special basic predicates for the considered systems to standard first-order and LTL connectives.

For a module, we need predicates to describe its local state, the input and output values and the respective adaptation counterparts. Therefore, equality and the less-than-or-equal relation over terms build from the relevant variables are employed. Furthermore, the configuration currently used is described by the predicate $\mathit{use\_conf}_{t_2}(t_1)$ which is true if the module denoted by term t_2 uses the configuration denoted by t_1 in the current state. On system level, we have predicates in order to speak about the connections between output and input variables implemented by the predicates $\mathit{is\_conn}_d(x_1, x_2)$ and $\mathit{is\_conn}_a(x_1, x_2)$ which are true if there is a functional or an adaptive connection between x_1 and x_2.

Definition 5 (Linear $\mathcal{L}_{SAS}$). *The grammar of linear $\mathcal{L}_{SAS}$ is defined as follows:*

$$\begin{aligned}
t \in \mathit{Terms} &::= x \in \mathit{Var} \mid v \in \mathit{Val} \mid f(t_1, \ldots, t_n) \\
a \in \mathit{Atoms} &::= t_1 = t_2 \mid t_1 \leq t_2 \mid \mathit{is\_conn}_{[a|d]}(x_1, x_2) \mid \mathit{use\_conf}_{t_2}(t_1) \\
\varphi \in \mathit{StFmlae} &::= \mathit{true} \mid a \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \rightarrow \varphi_2 \mid \exists x.\ \varphi \mid \forall x.\ \varphi \\
\psi \in \mathit{Fmlae} &::= \varphi \mid \mathbf{X}\ \psi \mid \ \mathbf{F}\ \psi \mid \ \mathbf{G}\ \psi \mid \psi_1\ \mathbf{U}\ \psi_2
\end{aligned}$$

A Formula $\varphi \in \mathcal{L}_{SAS}$ is interpreted over a path π where π is an infinite sequence of global states $\pi = \sigma^0, \sigma^1, \ldots$ which forms a system trace as defined in the previous section. We denote that a system trace of a synchronous adaptive system SAS $\pi \in \mathit{Runs}(SAS)$ is a model for a formula φ by $\pi \models_{SAS} \varphi$.

The interpretation of the temporal and first-order formulae complies to standard first-order LTL semantics [3]. Terms are evaluated by simply extending the variable assignments of a state σ to $\hat{\sigma}$. Equality and less-than-or-equal relation are interpreted standardly. The predicates $\mathit{is\_conn}_a(x_1, x_2)$ and $\mathit{is\_conn}_d(x_1, x_2)$ are valid if there is a connection via the respective connection functions. For reasoning purposes only, we enhance the local state of module i with an additional state variable config_i which captures the configuration that is used in this state. Its value is determined during the global transition $\sigma \rightarrow \sigma'$ by using the *adapt_trigger* function as defined in Definition 1 which selects the applicable configuration.

$$\sigma'(\mathit{config}_i) = \mathit{adapt\_trigger}_i(s'_i|_{in \,\cup\, adapt\_in} \cup s_i|_{loc \,\cup\, adapt\_loc})$$

Then, we are able to define the predicate *use_conf* over a state σ by $\mathit{use\_conf}_{t_2}(t_1) \equiv \mathit{true}$ iff $\sigma(\mathit{config}_{\hat{\sigma}(t_2)}) = \hat{\sigma}(t_1)$.

Furthermore, the boolean connectives are interpreted standardly. The next operator, $\mathbf{X}\,\varphi$, determines that φ is true in the next state, i.e. over the path π^1 which is the state sequence $\sigma^1, \sigma^2 \ldots$. A formula is globally true, $\mathbf{G}\,\varphi$, iff for all $i \geq 0$, φ holds over π^i, the path π starting in the i-th state. The formula $\mathbf{F}\,\varphi$ is true is there exists $i \geq 0$ such that $\pi^i \models_{SAS} \varphi$. The until operator $\varphi_1 \,\mathbf{U}\, \varphi_2$ denotes that there exists $j \geq 0$ such that $\pi^j \models_{SAS} \varphi_2$ and for all $0 \leq i < j$, $\pi^i \models_{SAS} \varphi_1$. We say that a formula $\varphi \in \mathcal{L}_{SAS}$ is valid for an SAS if $\pi \models_{SAS} \varphi$ for all paths $\pi \in \mathit{Runs}(SAS)$ and that it is satisfiable if there exists $\pi \in \mathit{Runs}(SAS)$ such that $\pi \models_{SAS} \varphi$.

5 Towards Modular Verification using Abstraction

Having defined a specification logic on top of the formal model we are now able to formally verify properties specified in $\mathcal{L}_{SAS}$. This verification process should incorporate automatic verification techniques such as model checking whenever possible. For an intuition how the proposed framework can be applied we consider the running example of the sensor quality adaptation as described in Section 3.1. The safety property to be shown is that the quality of the output at the actuator is never below 50 for more than 2 subsequent cycles. Otherwise, the actuator may break down causing the system to enter a dangerous situation. In $\mathcal{L}_{SAS}$, this property can be expressed by the formula φ that is required to hold for all paths $\pi \in \mathit{Runs}(SAS)$.

$$\varphi = \mathbf{G}\,\neg(\mathit{confidence}_{out} \leq 50 \wedge \mathbf{X}\,\mathit{confidence}_{out} \leq 50 \wedge \mathbf{X}\,\mathbf{X}\,\mathit{confidence}_{out} \leq 50)$$

For verification we proceed as follows. Firstly, we abstract the unbounded integer domain of the confidence level to three discrete values *low*, *med* (for medium) and *high*. This is necessary because automatic model checking techniques to be applied later can in general only deal with finite state systems. The abstraction has to preserve the properties of the concrete system, i.e. if the abstract property holds over the abstract system, also the concrete property holds over the concrete system. For our example, we can construct an abstract $SAS^{\#}$ along the lines of [2]. We use the following surjection h for mapping concrete confidence integer values to abstract values:

$$h(\mathit{confidence}) = \begin{cases} \mathit{low} \text{ if } \mathit{confidence} \leq 50 \\ \mathit{med} \text{ if } 50 < \mathit{confidence} \leq 100 \\ \mathit{high} \text{ if } \mathit{confidence} > 100 \end{cases}$$

For the paths of the abstract system $SAS^{\#}$ we have to ensure two conditions such that $SAS^{\#}$ approximates SAS and preserves its $\mathcal{L}_{SAS}$ properties. Firstly, the set of concrete initial states must be mapped to the set of abstract initials states. Secondly, the concrete transition relation must be contained in the abstract transition relation. In our example, this means that we must abstract all

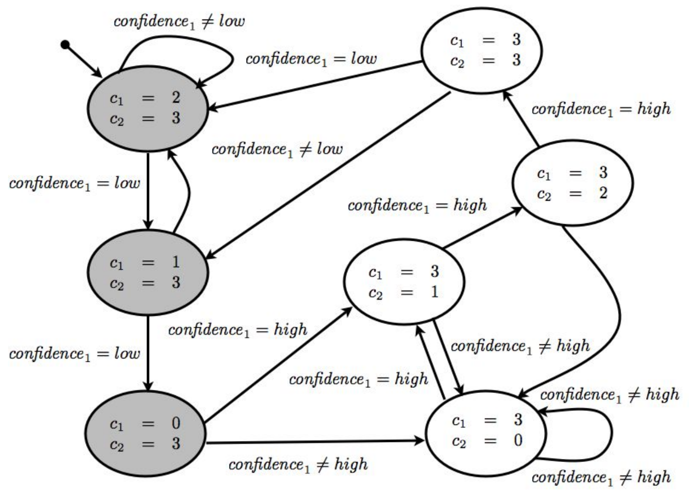

Fig. 3. State Transition Diagram for the Abstract Adaptation Behaviour of Module 1

conditions in configuration guards, adaptation next state and adaptation output functions that depend on $confidence_1$, $confidence_2$, $confidence_{12}$ and $confidence_3$ by the corresponding expressions using the abstract values *low*, *med* and *high*. For an example consider the abstracted transition diagram for module 1 in Figure 3. Additionally, the functions for calculating the confidence outputs have to be abstracted. This is easy for module 1 since it just propagates the relevant confidence already abstract in the abstract system. For module 2, the *adapt_next_out* function is defined as $confidence_{out} := min\{confidence_{12}, confidence_3\}$. Here, we have to give an abstract minimum function $min^\#$ which reflects the intuitive ordering that *low* is smaller than *med* which is smaller than *high*. The abstract property reads as follows: $\varphi^\# =$

$$\mathbf{G}\,\neg(confidence_{out} = low \,\wedge\, \mathbf{X}\,confidence_{out} = low \,\wedge\, \mathbf{X}\,\mathbf{X}\,confidence_{out} = low)$$

Verification of the abstract property $\varphi^\#$ over the abstract system $SAS^\#$ immediately implies validity of φ over SAS by construction of the abstraction using the results of [2].

Secondly, we modularly verify the abstract property over the abstract system. Therefore, we decompose the global property into two local properties over the two modules. From their validity can infer validity of the overall system property. The global property $\varphi^\#$ can be decomposed as follows. For module 2, we use the definition of the *adapt_next_out* function and show $\varphi_2^\# =$ (where c is used as abbreviation for *confidence*)

$$\mathbf{G}\,\neg(min^\#\{c_{12}, c_3\} = low \wedge \mathbf{X}\,min^\#\{c_{12}, c_3\} = low \wedge \mathbf{X}\,\mathbf{X}\,min^\#\{c_{12}, c_3\} = low)$$

By assumption on sensor 3 that its confidence is always greater than 50 or greater than *low* this property boils down to $\varphi_2^\# =$

$$\mathbf{G}\,\neg(confidence_{12} = low \wedge \mathbf{X}\,confidence_{12} = low \wedge \mathbf{X}\,\mathbf{X}\,confidence_{12} = low)$$

This is actually a property over module 1. So it suffices to prove $\varphi_1^\# \equiv \varphi_2^\#$ over module 1. We can enter this property together with the abstract module description into a model checker, for instance [9]. This will explore all paths of the abstract system and return the result that for all paths π of the abstracted module 1, $\pi \models_{m^\#_1} \varphi_1^\#$. This can also be seen in the abstract transition graph as depicted in Figure 3. If the confidence of sensor 1 is *low* for 2 subsequent cycles the system adapts to use sensor 2. Sensor 2 by assumption has a confidence of greater than 50 or in the abstract greater than low. So, the property $\varphi_1^\#$ holds on all execution paths. As a consequence, we can conclude by combining the results of abstraction and modularity that the example *SAS* satisfies the initial property φ.

6 Conclusion and Future Work

In this paper, we have introduced a formal semantic-based framework to model, specify and verify the functional and the adaptation behaviour of synchronous adaptive systems. The modelling framework separates functional and adaptive behaviour in order to reduce the design complexity and to allow modular reasoning about both aspects independently but also in combination. We have shown how to apply this framework for the verification of a safety property by the example of a sensor quality adaptation system.

As we have observed in the running example, modularity combined with abstraction reduces the complexity of sub-proof goals necessary to infer the desired overall system property. For these sub-goals, model checking algorithms such as [9] become efficiently applicable. Hence, for future work, we want to further investigate the use of modular verification in combination with abstraction mechanisms. In this direction, we want to integrate an automatic theorem prover dealing with modularity and abstraction with automatic model checking methods. Furthermore, we plan to equip our modelling framework with means for expressing hierarchy in order to be able to compose complex systems from a number of subsystems and to exploit this hierarchy for verification. In addition to that, we want to implement a translation from UML-like models of synchronous adaptive systems in the GME [7] framework to SAS models in order to provide GME models with a firm semantic basis and to make our approach end-user compatible by a graphical modelling front end.

References

1. J.S. Bradbury, J.R. Cordy, J. Dingel, and M. Wermelinger. A survey of self-management in dynamic software architecture specifications. In *Proc. of the International Workshop on Self-Managed Systems (WOSS'04)*, 2004.
2. E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1512–1542, 1994.

3. E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume B: Formal Models and Semantics*, pages 995–1072. Elsevier, Amsterdam, 1990.
4. W. Gilani, N. Naqvi, and O. Spinczyk. On adaptable middleware product lines. In *Proc. of 3rd Workshop an Adaptive and Reflective Middleware*, page 207213, 2004.
5. O. Gonzalez, H. Shrikumar, J. Stankovic, and K. Ramamritham. Adaptive fault-tolerance and graceful degradation under dynamic hard real-time scheduling. In *Proc. of IEEE Real-Time Systems Symposium (RTSS)*, pages 79–89, 1997.
6. S.S. Kulkarni and K.N. Biyani. Correctness of component-based adaptation. In *Proc. of Intl. Symposium on on Component Based Software Engineering*, pages 48–58, 2004.
7. A. Ledeczi and al. The Generic Modeling Environment. In *Proc. of IEEE International Workshop on Intelligent Signal Processing (WISP)*, 2001.
8. O. Marin, M. Bertier, and P. Sens. DARX - a framework for the fault tolerant support of agent software. In *Proc. of IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 406–418, 2003.
9. K. Schneider and T. Schuele. Averest: Specification, verification, and implementation of reactive systems. In *Proc. of Conference on Application of Concurrency to System Design (ACSD)*, 2005.
10. K. Schneider, T. Schuele, and M. Trapp. Verifying the adaptation behavior of embedded systems. In *Proc. of Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, 2006.
11. J. Zhang and B.H.C. Cheng. Specifying adaptation semantics. In *Proc. of ICSE 2005 Workshop on Architecting Dependable Systems (WADS 2005)*, pages 1–7, 2005.
12. J. Zhang and B.H.C. Cheng. Model-based development of dynamically adaptive software. In *Proc. of the International Conference on Software Engineering (ICSE'06)*, 2006.