EAI Endorsed Transactions
on Self-Adaptive Systems
Research Article ICST.ORG

Modular Design and Verification of Distributed Adaptive Real-Time Systems Based on Refinements and Abstractions ★

Thomas Göthel*, Verena Klös, Björn Bartels

Technische Universität Berlin
Department of Software Engineering and Theoretical Computer Science

Abstract

A promising way to cope with complexity in verifying large systems is to perform modular verification where components are verified separately. However, in the context of adaptive systems, it is difficult to apply this principle because adaptation behaviour and functional behaviour are often intertwined. In this paper, we present and apply a design pattern for distributed adaptive real-time systems using the process calculus Timed CSP. Our pattern explicitly differentiates between functional data and adaptive control data and thereby allows for a strict separation of adaptation and functional components. We enable the modular verification of functional and adaptation behaviour, respectively, based on the notion of process refinement in Timed CSP. The verification of refinements is automated using industrial-strength proof tools. As the notion of refinement can also be used to justify abstractions, we furthermore enable abstraction-based verification, where a detailed system is abstracted to facilitate more efficient verification efforts. This is especially important in the industrial development of adaptive systems using languages like SystemC where a designer not necessarily applies fine-grained refinements, but implements larger parts of the functional and adaptation logic possibly at the same time. Therefore, we discuss how common refinements and abstractions from the context of Timed CSP can be used as a formal basis for refinements and abstractions in SystemC.

Keywords: Adaptive Systems, Modelling, Verification, Refinement, Abstraction, Timed CSP, SystemC

Received on 30 October 2014, accepted on 09 January 2015, published on 28 January 2015

doi: 10.4108/sas.1.1.e5

1. Introduction

Modern adaptive systems are distributed among different network nodes. One of the advantages of such distributed adaptive systems is their robustness, which must not be corrupted by single points of failures as provoked by centralized components. Thus, adaptation of the entire network's behaviour should be distributed as well. This means that adaptive components should be able to adapt both, their local behaviour and the behaviour of the overall network, for example by notifying other components. This, however, makes these systems very complex to design and analyse.

In this paper, we present a design pattern for distributed adaptive real-time systems. Our aim is threefold. First, we describe an architecture that serves as a general template for the formal design of adaptive systems. Second, we enable a strict separation of functional and adaptation behaviour. Third, due to this separation, we allow for modular refinement- and abstraction-based verification of adaptive systems.

In previous work [3], we considered the separated refinement-based verification of adaptation and functional behaviour in untimed adaptive systems. In this paper, we extend this approach by introducing timing dependencies and by additionally focussing on a strict distinction between functional data and control data following [5]. A functional component manipulates its functional data but its behaviour may be controlled by possibly dynamic control data that can only be changed by some corresponding adaptation component. The adaptation component gathers information from the functional component, which it uses for an analysis concerning whether or not adaptation is necessary. Then, a plan is created which results in a set of new control data,

★This article is an extended version of [11]. The main extensions lie in the discussion of refinement-based verification in SystemC (Section 4.4) and in the discussion of abstraction-based verification in Timed CSP and SystemC (Section 6).

*Corresponding author. Email: thomas.goethel@tu-berlin.de

which is finally set in the functional component or sent to another distributed adaptive component. We show how this idea can be modelled and used for refinement-based verification with Timed CSP in a modular and stepwise manner using automatic tool support. Timed CSP is an ideal choice for this because of its expressive semantics, its compositional notions of refinement, and its mature tool support.

As the notion of refinement can be directly used to justify abstractions formally, we present several abstractions in Timed CSP and discuss how they can be used for verification (of adaptive systems). The benefits of abstraction-based verification is that a specific set of abstractions can be applied to make verification of a particular property feasible. The corresponding abstract model is not necessarily constructed in a refinement-based development process. Thus, abstraction-based verification can be applied in a more goal-oriented way.

We discuss both, the refinement- and the abstraction-based approach, w.r.t. their applicability in SystemC. We consider SystemC because of its industrial importance in the design and implementation of embedded real-time systems. This lays the foundation for a well-founded verification approach for distributed adaptive real-time systems that are designed in SystemC in future work. Furthermore, this would have the advantage that the rather theoretical language of (Timed) CSP can be applied in practice by a system designer without the need of explicitly working with (Timed) CSP.

The rest of this paper is structured as follows. In Section 2, we briefly introduce the process calculus Timed CSP and the system level design language SystemC, which is widely used in industry. Then, we discuss related work in Section 3. In Section 4, we introduce our timed adaptive specification pattern and discuss its refinement and verification capabilities. Furthermore, we discuss how its verification capabilities can be transferred to the context of SystemC. We illustrate the benefits of our approach using an example in Section 5. In Section 6, we discuss abstractions that can be used in the context of our pattern and discuss how these abstractions can be used in the context of SystemC. Finally, we conclude the paper in Section 7 and give pointers to future work.

2. Background

In this section, we give a brief introduction to Timed CSP followed by a short introduction to SystemC.

2.1. Timed CSP

Timed CSP is a timed extension of the CSP (Communicating Sequential Processes) process calculus [28]. It enables the description and the compositional refinement-based verification of possibly infinite-state real-time systems. Process operators like *STOP*, *SKIP*, *Prefix* (`a -> P`), *Sequential Composition* (`P ; Q`), *External Choice* (`P [] Q`), *Internal Choice* (`P |~| Q`), *Parallel Composition* (`P [|A|] Q`), *Hiding* (`P \ A`), and special timed operators like `WAIT(t)` are used to describe systems.

The basic processes of (Timed) CSP are *STOP* and *SKIP*. While *STOP* cannot do anything (except letting time advance), *SKIP* can also successfully terminate. The process `a -> P` offers the environment the opportunity to synchronize on event `a` at some point and then behaves like `P`. The process `P ; Q` first behaves like `P` and, if it successfully terminates, then behaves like `Q`. The process `P [] Q` offers its environment a choice between `P` and `Q`, which is triggered by the first visible event in either `P` or `Q`. The process `P |~| Q` behaves like either `P` or `Q`, but the choice is resolved internally without the influence of the environment. The parallel composition `P [|A|] Q` requires `P` and `Q` to synchronize on each event `a` $\in$ `A`, but all other events of either `P` or `Q` are performed independently. The process `P \ A` executes the events in the set `A` internally, without synchronization with the environment; they can be thought of as being replaced by indistinguishable τ events. Finally, the `WAIT(t)` operator can let `t time` units advance before it behaves like `SKIP`.

A discretely-timed dialect of Timed CSP that is amenable to automatic model checking techniques is tock-CSP. Here, the passage of time is explicitly modelled using a distinguished event *tock*. In FDR3 [10], which is the standard tool for CSP, tock-CSP is supported via timed operators and the `prioritise` operator which can be used to give internal (τ) as well as other events priority over `tock`. This is necessary to preserve the notion of refinement and its compositional features from Timed CSP.

Refinement is usually considered in the semantical traces or failures model. The refinement $P \sqsubseteq_T Q$, for example, expresses that $traces(Q) \subseteq traces(P)$ where $traces(\_)$ denotes all finite traces of a process. Traces can be used to specify safety properties. In contrast, failures additionally record a set of refused events at the end of each trace and thereby allow for the specification of liveness properties. Failures refinement is written as $P \sqsubseteq_{SF} Q$. While traces refinement is relatively intuitive, failures refinement basically describes reduction of (internal) non-determinism. This comes from the fact that refused events are only recorded in stable states after a trace, i.e., states where no internal step is possible. For example, `P |~| Q` is refined by the process `P [] Q` with respect to the (stable) failures model. The most important point concerning the semantics of (Timed) CSP is compositionality. From the refinements $P \sqsubseteq P'$ and $Q \sqsubseteq Q'$ it follows that in any arbitrary composition also $P \otimes Q \sqsubseteq P' \otimes Q'$ holds, i.e., refinement can be shown component-wise. This enables modular

verification in (Timed) CSP, which we exploit in the context of our adaptive system pattern.

2.2. SystemC

SystemC [16], introduced by the Open SystemC Initiative (OSCI) in 1999, is a system level design language and a framework for HW/SW co-simulation. It is implemented as a C++ library and provides elements for the description of both hardware and software, and of communication and synchronization between concurrent processes. It allows modelling and execution of system designs on various levels of abstraction. A SystemC design consists of communicating processes, which are triggered by events and communicate through channels. Structural information is represented in terms of modules (computing units) and channels (communication units). SystemC also introduces an integer-valued time model with arbitrary time resolution. The execution of SystemC designs is controlled by a cooperative and non-preemptive scheduler. It controls the simulation time, the event notification, the execution of processes, and the update of primitive channels. To impose a partial order on simultaneous actions, SystemC supports the notion of delta-cycles. A delta-cycle consists of two phases: the evaluation of ready processes and the update of primitive channels. The order in which concurrent processes are executed is chosen non-deterministically. SystemC supports immediate, delta, and timed notification where immediate notified processes are triggered before the update of primitive channels and delta notified processes afterwards.

The semantics of SystemC is informally defined by the SystemC scheduler, but there also exist a variety of approaches that provide a formal semantics for subsets of SystemC (as summarised in the next section).

3. Related Work

Dynamic reconfiguration of systems is supported by the architecture description language (ADL) Dynamic Wright [2]. Reconfiguration of interacting components is modelled separately from steady-state behaviour in a central specification. Our work aims at supporting the stepwise construction of distributed adaptive systems in which adaptation is realised in a decentralised way.

The work in [1] provides a model-based development approach for adaptive embedded systems in which adaptation behaviour is strictly separated from functional behaviour. Verification properties are expressed in temporal logics and verified using theorem provers and model checkers. In contrast, our approach aims to support development processes for adaptive real-time systems with the powerful notion of refinement in Timed CSP and proof tools for automatic refinement checking.

In [18], CSP is used to model self-adaptive applications where nodes in a network learn from the behaviour of other nodes. Behavioural rules, which are used to adapt the individual behaviour, are described in terms of CSP processes and communicated between the nodes. In contrast, we focus on modelling entire (self-)adaptive systems using Timed CSP and verifying properties of them based on the notion of refinement.

In [30], a modular approach for model-checking adaptive systems is presented. Invariant properties are stated in the temporal logic A-LTL, an extension of LTL with an adapt-operator. Modular verification is performed by decomposing the system into submodules and by applying assume-guarantee reasoning. Assumptions and guarantees are computed automatically. In contrast to our work, neither refinement nor real-time aspects are considered.

Timed automata are used in [17] for modelling and verifying a decentralised adaptive system. Verification of safety, liveness, and timing properties is performed using the Uppaal model checker. In contrast, we focus on the stepwise development and modular verification.

In [21], a UML-based modelling language for untimed adaptive systems is presented. Based on its formal semantics, deadlock freedom and stability can be verified. Our work enables the stepwise development and furthermore the verification of general functional and adaptation properties in a timed setting.

In [3], we presented an approach for the specification and verification of untimed distributed adaptive systems in CSP. A main goal of that work was the separation of functional from adaptation behaviour. The application of that framework in [29] has shown that the high level of abstraction becomes problematic when supplementing the adaptive system model with functional behaviours. While functional and adaptation events and also respective system variables can be separated, it remains unclear how the interface can be modelled in a systematic manner. This drawback is addressed in this paper following IBM's MAPE approach. This allows for a more modular verification approach compared to our previous work. Furthermore, we introduce mechanisms to specify and verify timing behaviour.

Related Work on SystemC can be split into two groups. On the one hand, there are modelling approaches for adaptive systems in SystemC that do not consider formal verification and on the other hand, there are approaches for formal verification of SystemC designs that do not consider adaptivity. [9, 24, 27] provide modelling mechanisms for reconfigurability in SystemC. In [24, 27] additional libraries for SystemC providing elements for modelling of reconfigurable components are proposed. In [9] reconfiguration of hardware tasks is modelled using dynamic threads. ANDRES [14] provides a modelling framework for

adaptive heterogeneous embedded systems, based on three SystemC extensions. It supports multiple means of adaptivity. Adaptivity is modelled with a formally defined adaptive process, which changes its behaviour depending on signals on a special control channel. In [13], a refinement flow from such an abstract adaptive process into dynamically reconfigurable hardware is presented. These modelling approaches explicitly focus on adaptive systems but do neither consider distributed adaptive systems nor support formal verification.

There exist a lot of formal verification approaches for subsets of SystemC (e.g. [6, 12, 20]), but they consider neither adaptivity nor modular verification. In [25] a verification framework for SystemC that uses state reduction and abstraction techniques is proposed, but adaptivity is not considered.

4. Timed Adaptive System Pattern and Refinement-Based Verification

In this section, we introduce an abstract pattern for distributed adaptive real-time systems. It defines a general structure of such systems, which is amenable to modular refinement-based verification. In Figure 1, the overall architecture is illustrated. We consider adaptive systems consisting of a network of adaptive components (AC(i)) that communicate using events. Communication events are categorised, depending on their purpose, as either functional events (FE) or external adaptation events (EA). A single component can perform some computation (also depicted by the occurrence of an abstract FE event) or adapt its internal behaviour (IA) due to the violation of some (local) invariant. Note that such computations are represented abstractly by events that possibly take time and by possibly changing the state variables. Internal adaptation can also be triggered by an internal timeout (TO). Timeouts can, for example, be used to indicate that during a certain amount of time, certain functional events have not been communicated. If some internal adaptation takes place, other components can be triggered to adapt their behaviour accordingly using external adaptation events (EA). The environment interacts with the adaptive system using functional events (FE) only. As it might be necessary to restrict the behaviour of the environment, the process ENV can be used to constrain it.

In a model-driven development process, an abstract design is continuously refined until an implementation model is reached. To start with more abstract models offers the advantage that properties can be verified, whose verification would be too complex on more concrete levels. In the following, we explain how the pattern can be formally defined on an abstract level in Timed CSP and how it can be refined in a stepwise fashion. This enables verification of properties in a modular manner. The primary focus of the models is

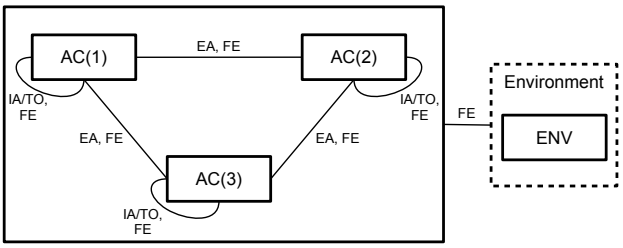

Figure 1. Architecture of Specifications

on the strict separation of functional behaviour and adaptation behaviour. Basing verification on the notion of refinement allows us to verify both of these aspects separately while leaving out concrete details of the respective other part. Using the real-time capabilities of Timed CSP and FDR3, we can particularly specify and verify timing dependencies in the functional and adaptation behaviours.

4.1. Abstract Model

In our approach, an adaptive system consists of a set of (distributed) adaptive components. Each such component consists of an adaptation component, a functional component, and, if necessary, a timer.

```
AdaptiveComponent(i) =
        (AC(i) [|{timeout}|] TIMER(i))
  [|union(FE(i),{|getData,setControlData|})|] FC(i)
```

The adaptation component checks whether adaptation of the functional component is necessary every `t(i)` time units. To this end, it implements IBM's MAPE (monitor, analyse, plan, execute) approach [15]. It gathers the data from the functional component using the `getData` event, analyse it, plan adaptation and execute the plan by setting the control data and possibly notifying other adaptive components. These steps are captured in the `CHECKADAPT` and `ADAPT` processes described below. The adaptation component can also be triggered by some external adaptation event (initiated by another adaptive component) or be notified that the timeout has elapsed. The timeout can for example be used to denote that during the last `timer(i)` time units no functional event took place (see `TIMER` below).

```
AC(i) =
     WAIT(t(i)) ; CHECKADAPT(i)
  [] ([] x:EA(i) @ x -> getData?d?cd -> ADAPT(i,x))
  [] timeout  -> getData?d?cd -> ADAPT(i,timeout)
```

The `CHECKADAPT` process gathers functional and control data from the functional component. Depending on violations of the local invariant, control data is adapted in the `ADAPT` process. On this abstract level, the invariant is not explicitly captured but possible violations are modelled via internal choices.

```
CHECKADAPT(i) = getData?d?cd ->
                    (|~| x:IA(i) @ x -> ADAPT(i,x)
                     |~| AC(i))
```

Adaptation takes some time `ta(i,x)`, depending on the component `i` in which adaptation takes place and depending on the cause of adaptation `x`. After the plan is created, the corresponding control data is set in the functional component and further adaptive components are notified using external adaptation EA events, which is realised in the `NotifyACs` process.

```
ADAPT(i,x) = WAIT(ta(i,x)) ;
             |~| cd : CD @ setControlData.cd ->
                           (NotifyACs(i,x) ; AC(i))
```

The timer keeps track of whether some functional event took place within the last `timer(i)` time units.

```
TIMER(i) = [] x:FE(i) -> TIMER(i)
           [] WAIT(timer(i)) ; timeout -> TIMER(i)
```

The functional component provides information about the internal data to the adaptation component. It also can obtain new control data from the adaptation component. On this level of abstraction, we abstract away state information using constructions based on internal choices. Furthermore, a functional component can communicate with other functional components or just manipulate its functional data using FE events, which may take some time `tf(fe)`.

```
FC(i) =    |~| (d,cd):{(d,cd) | d <- D , cd <- CD}
                    @ getData.d.cd -> FC(i)
        [] setControlData?cd' -> FC(i)
        [] |~| fe:FE(i) @ fe -> WAIT(tf(fe)); FC(i)
```

The abstract components have a far smaller state space than the refined components that we introduce in the following subsection. This especially enables the verification on the abstract level in reasonable time. The relatively complicated construction for coping with state information based on internal choices (e.g. `getData` in the functional component) is necessary to allow for later refinements in the failures model of CSP. It is certainly a radical way to leave out all of the state information here. However, it would be possible to keep at least a part of the state information.

4.2. Refined Model

In the abstract model described above, state information of the components is abstracted away. A refined model needs to make clear when the abstract actions actually take place. To do this in the context of CSP, non-determinism is usually reduced by using guarded deterministic choices ([]) instead of internal choices (|~|). For the adaptation component this means that the adaptation logic is refined by reducing non-determinism in CHECKADAPT and ADAPT. In the CHECKADAPT' subcomponent, the invariant is now explicitly modelled by the `g(i,d,cd,ia)` predicate. Note that CHECKADAPT' and ADAPT' now also depend on the functional (d) and control data (cd).

```
AC'(i) = WAIT(t) ; CHECKADAPT'(i)
      [] ([] x:EA @ x -> getData?d?cd
                        -> ADAPT'(i,d,cd,x))
      [] timeout  -> getData?d?cd
                  -> ADAPT'(i,d,cd,timeout)
```

```
CHECKADAPT'(i) = getData?d?cd ->
          ([] ia:IA(i) @ g(i,d,cd,ia) & ia
                          -> ADAPT'(i,d,cd,ia)
           [] else & none -> AC'(i))
```

```
ADAPT'(i,d,cd,x) = WAIT(ta(i,x)) ;
                   setControlData.f(i,d,cd,x) ->
                   NotifyACs'(i,d,cd,x) ; AC'(i)
```

The functional component no longer abstracts from the data, but makes use of it to implement the actual functional logic using guards (`gf(...)`), for example.

```
FC'(i,d,cd) = getData.d.cd -> FC'(i,d,cd)
     [] setControlData?cd' -> FC'(i,d,cd')
     [] ([] fe:FE(i) @ gf(i,d,cd,fe) & fe
                -> WAIT(tf(fe)); FC'(i,h(d,fe),cd))
```

In the next section, we explain the modular refinement and verification process in the context of the presented adaptive system pattern.

4.3. Proving Refinement

The aim of the described pattern is to facilitate the modular verification of adaptive real-time systems. The most abstract system model leaves out most of the details concerning adaptation and functional behaviour. Thus, the most abstract model is suited to verify properties, which focus neither on the adaptation behaviour nor the functional behaviour, e.g., abstract communication properties. By introducing more detailed adaptation or functional behaviour, we refine the abstract model to models that fulfil more detailed properties w.r.t. adaptation or functional behaviour, respectively. The key point is that often only the functional behaviour or the adaptation behaviour needs to be refined, not both at the same time.

A refinement-based approach for verification has two major advantages. First, we can verify functional and adaptation correctness separately (see Figure 2). On the most abstract level, the system is composed of functional components $FC(i)$ and adaptation components $AC(i)$. Both of these abstract kinds of components leave out most of the implementation details. When refining a functional component to $FC'(i)$ or an adaptation component to $AC'(i)$, we can verify functional and adaptation properties separately by leaving out details of the respective other components, which are not of interest for the respective property. This

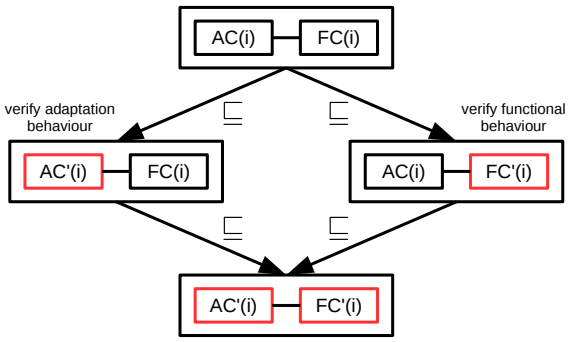

Figure 2. Refinement Strategy

kind of component-wise refinement is possible due to compositionality of (Timed) CSP refinement. For the composed system, we have $FC(i) \otimes AC(i) \sqsubseteq_{FD} FC(i)' \otimes AC(i)$ and $FC(i) \otimes AC(i) \sqsubseteq_{FD} FC(i) \otimes AC'(i)$. Furthermore, we have that $FC'(i) \otimes AC(i) \sqsubseteq_{FD} FC'(i) \otimes AC'(i)$ and $FC(i) \otimes AC'(i) \sqsubseteq_{FD} FC'(i) \otimes AC'(i)$. Thus, all properties that are valid on the partly refined models $FC'(i) \otimes AC(i)$ and $FC(i) \otimes AC'(i)$ remain valid in the refined model $FC'(i) \otimes AC'(i)$. The second advantage is related to the environment model. In CSP, a model is more abstract than another if it contains fewer constraints. This means that a refined system has fewer behaviours than the abstract one. Ideally, an adaptive system is verified with a most abstract or most unconstrained environment. However, this is almost never possible, especially not in the context of adaptive systems. Fortunately, refinement allows us to introduce necessary constraints to the environment.

4.4. Towards Refinement for SystemC

Our pattern from Section 4 is not restricted to Timed CSP, but can also be used to develop adaptive systems in design languages like SystemC. An adaptive SystemC model consists of (distributed) adaptive components, modelled as hierarchical modules that communicate using channels. Each such module contains a functional and an adaptation module, which realises the MAPE approach. SystemC supports a stepwise design process starting with an abstract specification, which is refined by adding functional and communication details as well as accurate timing.

Refinement proofs allow for transferring verification results from an abstract design to a refined one. For proving refinement, a formal semantics is necessary. However, the SystemC semantics is only informally defined. There exist several approaches that define formal semantics to enable verification (see Section 3). However, as far as we know, there are no approaches for proving refinement between SystemC designs directly.

One possibility to prove trace refinement between two SystemC designs is to combine an approach that defines a formal semantics for SystemC via transformation to UPPAAL Timed Automata (a timed extension of finite state machines) [12] with an approach for proving trace-based refinement for timed automata [7, 8]. Using this idea, trace refinement can be shown by transforming both, the abstract and the refined SystemC design to timed automata and verify whether there exists a trace refinement between them.

The work of [12] supports a large subset of SystemC including the SystemC transaction level modelling standard TLM [22] and the most important memory related operations [23] and enables model checking with the UPPAAL model checker. Therefore, it is well suited for the formal verification of SystemC designs. Trace-based refinement checking for timed automata can be done with the notion of timed simulation for externally observable traces (hiding local synchronisation) [4, 7]. [4] checks refinement between two cottbus timed automata using a fixpoint algorithm on the set of reachable states of the composition of the two automata. In [7], refinement checking considering timed traces of Timed Input/Output Automata is done by solving a safety timed game. This approach is implemented in the ECDAR [8] tool, an extension of UPPAAL-TIGA (timed games).

5. Example

In this section, we present a simple adaptive system to illustrate the main ideas of the adaptive system pattern from the previous section. It consists of two adaptive components: a light dimmer and a daylight sensor. When the daylight sensor recognises a change in light intensity that stays stable for a certain amount of time, the dimmer is notified that it should adapt to the new situation by changing the dim intensity. Furthermore, the dimmer can be adjusted manually. This represents the actual functional behaviour of the dimmer. On the abstract level, we omit details concerning the state information within the components. This means that all choices, which depend on the state information are realised by internal choices.

As the process `DimmerFC_0` below shows, the dimmer is adjusted manually using the `higher` and `lower` events. Furthermore, the dim intensity can be set using the `setGoal` event leading to an automatic adjustment phase thereafter. Finally, the current dim value can be queried. The `obs` event is used as an observation event for later verification only.

```
DimmerFC_0(y) =  higher -> obs?x -> DimmerFC_0(y)
              [] lower -> obs?x -> DimmerFC_0(y)
              [] setGoal?ny -> DimmerFC_0(9)
              [] (y>0 & (adjust -> DimmerFC_0(y-1)
                        |~| DimmerFC_0(0)))
              [] y==0 & obs?x -> DimmerFC_0(-1)
              [] getCurrent?x -> DimmerFC_0(y)
```

The corresponding adaptation component can be notified that the intensity of the surrounding light has changed such that it subsequently adapts the behaviour of the functional component.

```
DimmerAC_0 = newIntensity?y
          -> getCurrent?x
          -> (DimmerAC_0
              |~| setGoal?x -> DimmerAC_0)

AdaptiveComponent1_00 = DimmerAC_0
     [| {| getCurrent, setGoal |} |] DimmerFC_0(-1)
```

The light sensor recognises the daylight intensity. If it remains stable for 5 time units, it is checked whether the dimmer needs to be notified using the `newIntensity` event. On this abstract level, details of the check are hidden through an internal choice.

```
LightSensorTimer =
     WAIT(5) ; timeout -> LightSensorTimer
  [] light?y -> LightSensorTimer

LightSensorAC_0 =
              timeout
           -> getIntensity?y
           -> (newIntensity?x -> LightSensorAC_0
               |~| LightSensorAC_0)

LightSensorFC_0 = light?x -> LightSensorFC_0
                [] getIntensity?x -> LightSensorFC_0

AdaptiveComponent2_00 =
  ((LightSensorAC_0
     [| {timeout} |] LightSensorTimer)\{timeout})
     [| {| getIntensity |} |] LightSensorFC_0
```

The environment model formalises the restriction that the system is interacted with at most once a second. This is clearly a severe restriction but eases presentation here. Finally, the system model assembles the adaptive components and the environment model according to the architecture given by our adaptive pattern.

```
ENV = WAIT(1);(    light?y->ENV
               [] higher->ENV
               [] lower->ENV)

System_abs_0 = ((AdaptiveComponent1_00
                 [|{|newIntensity|}|]
                AdaptiveComponent2_00)
                 [|{|light, higher, lower|}|]
  ENV)\{| newIntensity, getCurrent, getIntensity |}
```

We have modelled three safety properties as CSP processes, which can be verified using trace refinement. The first property states that two consecutive `setGoal` events always occur with different values. The second one states that there is a delay of at least 4 time units between consecutive `setGoal` events. Finally, the third property states that the dimmer is adjusted gradually. The dim value before and after setting it can differ by two at most.

```
Prop1 = ([] x:diff(Events,{|setGoal|})@ x -> Prop1)
        [] setGoal?x -> Prop1_help(x)

Prop1_help(x) =
  ([] e:diff(Events,{|setGoal,higher,lower|})
          @ e -> Prop1_help(x))
   [] setGoal?y:{y | y <- DIMINTENSITY , y != x}
          -> Prop1_help(y)
   [] higher -> Prop1
   [] lower -> Prop1
```

```
Prop2 = ([] x:diff(Events,{|setGoal|})@ x -> Prop2)
        [] setGoal?x -> Prop2_help(0)

Prop2_help(x) =
  ([] e:diff(Events,{|setGoal,tock|})
          @ e -> Prop2_help(x))
   [] tock -> Prop2_help(addbound(x,1,10))
   [] x>=4 & setGoal?x -> Prop2
```

```
Prop3 = ([] x:diff(Events,{|obs|})@ x -> Prop3)
        [] obs?x -> Prop3_help(x)

Prop3_help(x) =
 ([] e:diff(Events,{|obs|}) @ e -> Prop3_help(x))
  [] obs?y:{addbound(x,-2,10), addbound(x,-1,10),x,
            addbound(x,1,10), addbound(x,2,10)}
            -> Prop3_help(y)
```

These properties are not valid in the abstract model above. As all three properties are concerned with the adaptation behaviour of the two components, we first refine the adaptation mechanisms accordingly.

```
DimmerAC_1 =
     newIntensity?y
  -> getCurrent?x
  -> if (x-y < 0) or (x-y > 9) then DimmerAC_1
                else setGoal.(x-y) -> DimmerAC_1

LightSensorAC_1(x) =
  timeout -> getIntensity?y
  -> if (y!=x) then newIntensity.lDiff(x,y)
                    -> LightSensorAC_1(y)
                else    LightSensorAC_1(x)
```

The definitions of the adaptation component above are updated with these refined parts accordingly (taking 0 as the initial value for `x`). The corresponding new system description `System_abs_1` is sufficiently refined to show the second property using FDR3. Now, we have two prove obligations: First, it has to be shown that `System_abs_1` is indeed a refinement of `System_abs_0` and that the second property holds.

For the refinement check, we exploit compositionality of CSP. Instead of showing refinement of the overall system, we show it component-wise.

```
assert AdaptiveComponent1_00
       [FD= AdaptiveComponent1_10
assert AdaptiveComponent2_00
       [FD= AdaptiveComponent2_10
```

For showing the second property on the (partly) refined system, we need to prioritise internal events

over tock in the system model. Furthermore, we specify that the setGoal and obs events are urgent but visible to be able to express properties using these events.

```
assert P2 [T= prio(System_abs_1,
                   <{|setGoal,obs|},{tock}>)
```

The first and the third property do not hold on this model because they also depend on the functional behaviour of the dimmer. So, we also refine the DimmerFC component.

```
DimmerFC_1(x,y) =
   x<9 & higher -> obs.(x+1) -> DimmerFC_1(x+1,-1)
[] x>0 & lower -> obs.(x-1) -> DimmerFC_1(x-1,-1)
[] setGoal?ny -> DimmerFC_1(x,ny)
[] y>=0 and y>x & adjust -> DimmerFC_1(x+1,y)
[] y>=0 and x>y & adjust -> DimmerFC_1(x-1,y)
[] y>=0 and x==y & obs.x -> DimmerFC_1(x,-1)
[] getCurrent.x -> DimmerFC_1(x,y)
```

Again, we have to check refinement explicitly.

```
assert AdaptiveComponent1_10
       [FD= AdaptiveComponent1_11
```

With this refined version, we can finally show the first and the third property.

```
assert P1/P3 [T= prio(System_abs\_2,
                      <{|setGoal,obs|},{tock}>)
```

Note that the last property is not as obvious as it appears at first glance. If we did not have the environmental assumptions that there is a delay of at least one time unit between external events, a setGoal event could be arbitrarily delayed while higher and lower events have an effect on the dimmer.

For completeness, we also give the refined version of the functional component of the light sensor. Here, the last intensity value that has been recognised is memorised and can be given to the adaptation component accordingly.

```
LightSensorFC_1(x) =
          light?y -> LightSensorFC_1(y)
       [] getIntensity.x -> LightSensorFC_1(x)
```

In summary, we have shown that it is possible to verify the example above in a modular way by focussing especially on adaptation behaviour while abstracting from functional behaviour as much as possible. The advantages of this approach become even more striking when this pattern is applied to more complex adaptive systems. In future work, we plan to apply our approach to the case study of an adaptive multicore scheduler, which could not be verified conveniently [29] because there was no clear separation of functional and control data. By applying our pattern, we expect verification to be far more modular and more scalable. Furthermore, we would like to evaluate abstraction-based verification of such systems. In the next section, we briefly describe how this could be done.

6. Abstractions for Modular Verification of Adaptive Systems

In Sections 4 and 5, we have defined and illustrated a pattern for the design and analysis of distributed adaptive real-time systems. To be precise, we have considered the setting of a refinement-based development approach. Refinements preserve abstract properties and can automatically be checked. In practice, however, it is likely that the refinement steps are not that fine-grained but that there are rather large refinement steps. This might make it necessary to abstract from details in a design in order to be able to perform verification.

Abstractions are dual to refinements. Instead of adding details to some design, they abstract from details in order to make verification feasible. While in the refinement-based approach it is assumed that refinement is explicitly verified between models, abstractions should be correct in any case. The reason for this is that it might not be easily possible in concrete domains to verify refinement between models. For example, the domain of SystemC models allows for abstractions based on CSP-like refinement. However, to enable verification of concrete refinements in SystemC, a formal semantics supporting trace and failures refinement would be needed. Although results for trace refinements are partly available via a transformation to timed automata, sophisticated refinements like failures refinement are not yet supported.

In this section, we define abstractions in the CSP setting, which are safe in the sense that their constructions imply failures refinement. We briefly discuss how these abstractions can be used in the setting of SystemC to allow for state-space reduction and, thus, provide an efficient way to verification in an industrial system description language.

6.1. Safe Abstractions in CSP

In the following, we first sketch some comparatively simple abstractions in (Timed) CSP before we review two more sophisticated abstractions, based on data independence and timewise refinement. We argue that their combination can make (abstraction-based) verification of adaptive real-time systems feasible.

Control Abstraction. The basic idea in CSP-based refinement, especially in failures refinement, is to make decisions more concrete, i.e., to reduce non-determinism. We can clearly go the other way around by reducing concrete control by non-deterministic decisions. One source of concrete control is the use of guarded external choices. If we replace guarded external choices by internal choices, we obtain the required abstraction, which directly can be justified based on failures refinement. However, this abstraction is only valid, if it can be ensured that at least one alternative of the

guarded choices is available at each time. Otherwise a deadlock would be possible. As adaptive systems should be generally reactive systems and therefore non-terminating, such a deadlock can be considered as error in the design. Fortunately, it can be statically checked whether at least one guard is satisfied at each time by checking that $\bigwedge_{i \in X} g_i$ equals $\top$. With this condition, we can define two abstraction rules, which can be applied to the model of interest in a step-wise way.

```
    g_a & a -> P_a
[]  g_b & b -> P_b
==>
    a -> P_a
|~| b -> P_b
```

```
[] x:X @ g_x & x -> P_x
==>
|~| x:X @ x -> P_x
```

The presented abstraction captures the most obvious way to abstract (Timed) CSP models because it directly corresponds to the main principle of refinement in (Timed) CSP. In the following, we present abstractions that exploit the structure of the adaptive systems based on the presented pattern. We especially focus on the state information, which needs to be explicitly communicated to other components, and how it can be abstracted away in the (Timed) CSP context.

Observer Abstraction. In a state-based system, it is likely that state information is communicated in some way. As a first step to abstract away state-based information, we can hide exact state information that is communicated to some environment. To this end, we replace communication events by internal choices which range over the state information that is possibly communicated.

```
P(x) = ... obs.x -> ...
       ... comm.x -> ...
==>
P(x) = ... |~| x:X @ obs.x -> ...
       ... |~| x:X @ comm.x -> ...
```

If we use this abstraction until no exact state information is communicated, we can apply the next abstraction, which has the potential of heavily reducing the state space of (Timed) CSP models.

State Abstraction. If no event depends on (specific parts of) actual state information, (parts of the) state information itself can be abstracted away. In CSP, state can only be modelled explicitly within recursive process definitions which are parameterised over state information as parameter. As the denotational CSP semantics focus on observable behaviour only, parameters are only necessary if the behaviour depends on them. The previous abstractions allow us to eliminate these dependencies so that finally parameters of process definitions can be eliminated as well.

```
P(x) = ... |~| x:X @ obs.x -> ... P(x)
       ... |~| x:X @ comm.x -> ... P(x)

==>

P = ... |~| x:X @ obs.x -> ... P
    ... |~| x:X @ comm.x -> ... P
```

Above, we have sketched some possible abstractions that directly reverse refinement. This means that instead of reducing nondeterminism, we introduce nondeterminism in order to be able to reduce the state space of components.

In the following, we also review two famous abstractions that are available for (Timed) CSP. Together with the abstractions described above, they can provide a formal foundation of abstractions in SystemC as described in Section 6.2.

Communication and Functional Abstraction. The presented observer and state abstractions eliminate state dependencies, but still work with original data domains. This might result in large systems depending on the size of the underlying data type. For certain properties, however, it is sufficient if a smaller abstract data type is considered. These could be properties that do not depend on actual data, which is communicated or actual computation results of some function in the system. The formal basis for this idea is data independence, which has also been studied in the context of CSP [19]. Under certain (data independence) conditions on the usage of data, some possibly unbounded data type can be replaced by a finite data type, which is usually comparably small. Properties established on the system using the finite data type remain valid on the concrete system modelled with the possibly unbounded data type. These data independence conditions can be checked using only the syntactical structure of the CSP processes. Control abstraction together with data independence results have the potential to massively reduce the state space. While control abstraction allows for achieving actual data independence, the data independence results allow for verification based on a small range of data values.

Timing Abstraction. Often, one is not only interested in timing properties but also in untimed safety and liveness properties. To be able to abstract away detailed timing behaviour, the notion of timewise refinement [28] can be used as a correctness criterion. The idea is to formally relate untimed and real-time systems. To refine an untimed process to a real-time process, prefixes can be replaced by delayed prefixes and terminating $SKIP$ processes can be replaced by $WAIT$ processes. The idea is that the real-time system does not have to provide certain communication capabilities from the beginning but that it is sufficient if the system stabilises after some time such that

they are stably provided then. Timewise refinement is compositional for sequential processes. For concurrent processes this is, however, not necessarily the case. As a solution, it is possible to check the processes for "non-retraction" [28]. Recently, it has been shown how timewise refinement can be checked automatically using the FDR3 refinement checker [26]. By taking again a dual view on timewise refinement, abstractions can be defined by removing concrete timing information.

6.2. Transferring the Abstractions to SystemC

In this subsection, we illustrate how the safe abstractions from the last subsection can be used in the context of SystemC.

Control Abstraction. In CSP, control abstraction is used to replace concrete control by non-deterministic choices. In SystemC, such non-deterministic choices can be realized by replacing boolean conditions with a random boolean value and, in case of a switch statement, by replacing a state variable with a variable with random values.

Control abstraction introduces non-determinism and eliminates control dependencies on state variables enabling further abstractions to reduce the state space.

```
switch(state){
  case "a":
     foo(x);
     break;
  case "b":
     bar(x);
     break;
  case "c":
     bar(x);
     break;
}

if(condition) {...};
```

==>

```
int state = rand()%3;
switch(state){
  case 0:
     foo(x);
     break;
  case 1:
     bar(x);
     break;
  case 2:
     bar(x);
     break;
}

bool randcond = rand()%2;
if(randcond){...};
```

Observer Abstraction. Observer Abstraction in CSP hides state information that is communicated to the environment by introducing a non-deterministic choice on the possible communication events. In SystemC, this can be achieved in a similar way by introducing a random choice on the events or variable values.

These last two abstractions are not common in SystemC, whereas the next abstractions are often used in a top-down SystemC design process.

State Abstraction. In CSP, state information is represented by parameters of recursive process definitions and can be eliminated if the observable behaviour does not depend on them. To reduce the state space in SystemC, variables have to be eliminated or replaced by variables with a smaller range of possible values. This can be done easily for variables, which are never used or which can be directly derived from other variables. As SystemC has a stronger focus on internal computations than CSP, in addition to the observable behaviour, the influence of variables on internal computations have to be considered. To enable further abstractions those computations have to be abstracted.

Communication and Functional Abstraction. For verification of a property that does not depend on actual (communicated or calculated) data, data and computational abstractions can be used to abstract from the actual data. Data abstraction replaces actual data with abstract data to reduce the range of possible values. Data variables like integers can be replaced by enumerations on values representing equivalence classes of the original values. Complex data objects can be replaced by abstract data objects with less detail. Computational abstraction can be achieved by encapsulating internal computation and replacing it with a simpler computation or even with an abstract function result.

In SystemC, communication and computation are clearly separated (in channels and modules). Hence, they can be abstracted separately. A common abstraction is communication abstraction, where the communicated data is replaced with abstract data and the communication channel, implementing the communication protocol, is replaced by an abstract channel (e.g. message passing via fifo).

Timing Abstraction. An important abstraction to reduce the state space of SystemC designs is timing abstraction. This can be achieved by replacing timed notification with delta notification (i.e., replace `wait(t)` with `wait(SC_ZERO_TIME)`). With delta notification, no simulation time elapses, but the scheduler gets control for one delta cycle, enabling an update of all channels.

We have sketched how formal abstractions in CSP can be transferred to SystemC. In the case of communication, functional, and timing abstraction the transfer showed similarities between common abstractions in both worlds. Due to the fact, that both languages have similar principles (e.g., concurrent processes, event-based synchronisation), a transfer of formal abstractions in Timed CSP to SystemC is possible. In the following we illustrate how these abstractions can be used as systematic abstractions to enable abstraction-based modular verification.

6.3. Abstraction-Based Modular Verification of Adaptive Systems

For modular verification, our goal is to use the conceptual separation between adaptation and functional logic, as well as the modular structure of the overall system. The idea is to split the verification properties into sets of properties concerning the adaptation logic, concerning the functional logic, and concerning the interaction between components.

To verify a functional component (FC), we want to abstract the adaptation component (AC) concerning the observable behaviour at the interface between FC and AC. Expecting the AC to implement a MAPE-loop, only the monitoring and executing parts need to be verified. Therefore, we want to replace the AC with an abstract component, which implements the interface and communicates any control data to the FC in order to adjust the functional logic.

Respectively, we want to abstract the FC concerning the interface behaviour to verify the AC. Here, we can abstract from the functional behaviour for the monitoring and the analysing parts of the AC and use an abstract FC component communicating any control data to the AC. This approach can be used to verify properties like *t time units after a change in the control data (representing the state of FC), the AC receives the updated control data* or *The AC detects an invariant violation after at most t time units*.

Considering the planning and executing parts of the AC, we want to verify properties like *t time units after an invariant violation occurs, the invariant holds again*. Therefore, we cannot abstract away the functional behaviour completely, but have to consider the part that is affected by an adaptation and responsible for reestablishing the invariant.

In this section, we have presented several abstractions in CSP and corresponding abstractions in SystemC. Furthermore, we have illustrated our vision of how these abstractions can be used for the abstraction-based modular verification of timed adaptive systems.

7. Conclusion and Future Work

In this paper, we have presented a design pattern that supports the modular design and verification of distributed adaptive real-time systems. In particular, it clarifies how data is processed and communicated within the individual components of an distributed adaptive system. Adaptation is achieved in a decentralised fashion by the cooperation of the individual components. Moreover, we have demonstrated how timing dependencies can be modelled and analysed using a refinement-based approach. Using an example, we have shown how the approach facilitates the stepwise development of distributed adaptive real-time systems and helps to cope with the complexity of such systems by providing automated verification methods. We have also considered abstractions that can be based on the notion of Timed CSP refinement. This way, goal-oriented abstractions can be built that allow for verification of more complex systems. For both, the refinement- and the abstraction-based approach, we have discussed how they can be transferred to SystemC. Being an industrially widely used system description language, this provides the basis for practical verification of distributed adaptive real-time systems.

In future work, we plan to apply our approach to an adaptive multicore system, which was previously only incompletely verified [29] using Timed CSP because of limited scalability due to not strictly separating functional from adaptation behaviour. As another piece of work, we want to analyse whether we can exploit the compositional structure of systems in our approach to enable runtime verification. This would especially enable the integration of more flexible adaptation strategies at design time such that the system could apply the correct strategies at runtime while preserving functional and adaptation correctness.

Furthermore, we are interested in applying our ideas to the practical verification of adaptive real-time systems in SystemC. We are currently working on the implementation of a case study concerning an adaptive automotive emergency break system. In a first step, we plan to formalise our proposed abstractions and to formally relate them to the SystemC semantics.

References

[1] Adler, R., Schaefer, I., Schüle, T. and Vecchié, E. (2007) From model-based design to formal verification of adaptive embedded systems. In *Formal Methods and Software Engineering, 9th International Conference on Formal Engineering Methods (ICFEM)*: 76–95.

[2] Allen, R., Douence, R. and Garlan, D. (1998) Specifying and analyzing dynamic software architectures. In *Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE)*: 21–37.

[3] Bartels, B. and Kleine, M. (2011) A CSP-based framework for the specification, verification, and implementation of adaptive systems. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)* (ACM): 158–167.

[4] Beyer, D. (2001) Efficient reachability analysis and refinement checking of timed automata using BDDs. In *Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods* (Springer): 86–91.

[5] Bruni, R., Corradini, A., Gadducci, F., Lluch-Lafuente, A. and Vandin, A. (2012) A conceptual framework for adaptation. In *Fundamental Approaches to Software Engineering (FASE)* (Springer): 240–254.

[6] Cimatti, A., Narasamdya, I. and Roveri, M. (2013) Software model checking SystemC. *IEEE Trans. on CAD of Integrated Circuits and Systems* **32**(5): 774–787.

[7] David, A., Larsen, K.G., Legay, A., Nyman, U. and Wasowski, A. (2010) Timed I/O automata: a complete specification theory for real-time systems. In *Proceedings of the 13th ACM international conference on Hybrid systems: computation and control* (ACM): 91–100.

[8] David, A., Larsen, K.G., Legay, A., Nyman, U. and Wąsowski, A. (2010) ECDAR: An environment for compositional design and analysis of real time systems.

In *Proceedings of the 8th International Conference on Automated Technology for Verification and Analysis (ATVA)* (Springer): 365–370.

[9] Duhem, F., Muller, F. and Lorenzini, P. (2011) Methodology for designing partially reconfigurable systems using transaction-level modeling. In *Design and Architectures for Signal and Image Processing (DASIP)* (IEEE): 1–7.

[10] Gibson-Robinson, T., Armstrong, P., Boulgakov, A. and Roscoe, A. (2014) FDR3 — a modern refinement checker for CSP. In Ábrahám, E. and Havelund, K. [eds.] *Tools and Algorithms for the Construction and Analysis of Systems* (Springer Berlin Heidelberg), *Lecture Notes in Computer Science* **8413**, 187–201.

[11] Göthel, T. and Bartels, B. (2014) Modular design and verification of distributed adaptive real-time systems. In *Second International Workshop on Formal Methods for Self-Adaptive Systems (FMSAS 2014)*. Accepted, to be published.

[12] Herber, P., Fellmuth, J. and Glesner, S. (2008) Model checking SystemC designs using timed automata. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)* (ACM): 131–136.

[13] Herrera, F., Villar, E. and Hartmann, P.A. (2011) SystemC refinement of abstract adaptive processes for implementation into dynamically reconfigurable hardware. In *Specification and Design Languages (FDL)* (IEEE): 1–8.

[14] Herrholz, A., Oppenheimer, E., Hartmann, P.A., Schallenberg, A., Nebel, W., Grimm, C., Damm, M. *et al.* (2007) The ANDRES project: analysis and design of run-time reconfigurable, heterogeneous systems. In *Field Programmable Logic and Applications* (IEEE): 396–401.

[15] IBM Corp. (2004) *An architectural blueprint for autonomic computing* (USA: IBM Corp.). URL `www-3.ibm.com/autonomic/pdfs/ACBP2_2004-10-04.pdf`.

[16] IEEE Standards Association (2005) IEEE Std. 1666–2005, Open SystemC Language Reference Manual.

[17] Iftikhar, M.U. and Weyns, D. (2012) A case study on formal verification of self-adaptive behaviors in a decentralized system. In Kokash, N. and Ravara, A. [eds.] *FOCLASA*, *EPTCS* **91**: 45–62.

[18] Jaskó, S., Simon, G., Tarnay, K., Dulai, T. and Muhi, D. (2009) CSP-based modelling for self-adaptive applications. In *Infocommunications Journal*, **LVIV**: 14–21.

[19] Lazić, R. (1999) *A Semantic Study of Data Independence with Applications to Model Checking*. Ph.D. thesis, Oxford University.

[20] Le, H.M., Grosse, D., Herdt, V. and Drechsler, R. (2013) Verifying SystemC using an intermediate verification language and symbolic simulation. In *Proceedings of the 50th Annual Design Automation Conference (DAC)* (ACM): 116:1–116:6.

[21] Luckey, M. and Engels, G. (2013) High-quality specification of self-adaptive software systems. In *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)* (IEEE): 143–152.

[22] Pockrandt, M., Herber, P. and Glesner, S. (2011) Model checking a SystemC/TLM design of the AMBA AHB protocol. In *IEEE/ACM Symposium on Embedded Systems For Real-time Multimedia (ESTIMedia)* (IEEE): 66–75.

[23] Pockrandt, M., Herber, P., Klös, V. and Glesner, S. (2013) Model checking memory-related properties of hardware/software co-designs. In *Embedded Systems: Design, Analysis and Verification. Proceedings of the International Embedded Systems Symposium (IESS)*: 92–103.

[24] Raabe, A., Hartmann, P.A. and Anlauf, J.K. (2008) ReChannel: Describing and simulating reconfigurable hardware in SystemC. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* **13**(1): 15.

[25] Razavi, N., Behjati, R., Sabouri, H., Khamespanah, E., Shali, A. and Sirjani, M. (2010) Sysfier: Actor-based formal verification of SystemC. *ACM Trans. Embedded Comput. Syst.* **10**(2): 19.

[26] Roscoe, A. (2013) The automated verification of timewise refinement. *First Open EIT ICT Labs Workshop on Cyber-Physical Systems Engineering* .

[27] Schallenberg, A., Nebel, W., Herrholz, A., Hartmann, P.A. and Oppenheimer, F. (2009) OSSS+R: A framework for application level modelling and synthesis of reconfigurable systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)* (IEEE): 970–975.

[28] Schneider, S. (1999) *Concurrent and Real Time Systems: The CSP Approach* (John Wiley & Sons, Inc.).

[29] Schwarze, M. (2013) *Modeling and verification of adaptive systems using Timed CSP*. Master thesis, Technische Universität Berlin.

[30] Zhang, J., Goldsby, H.J. and Cheng, B.H. (2009) Modular verification of dynamically adaptive systems. In *Proceedings of the 8th ACM International Conference on Aspect-oriented Software Development (AOSD)* (ACM): 161–172.