Modular Design and Verification of Distributed Adaptive Real-Time Systems

Thomas Göthel(✉) and Björn Bartels

Technische Universität Berlin, Berlin, Germany
{thomas.goethel,bjoern.bartels}@tu-berlin.de

Abstract. We present and apply a design pattern for distributed adaptive real-time systems using the process calculus Timed CSP. It provides a structured modelling approach that is able to cope with the complexity of distributed adaptive real-time systems caused by the interplay of external stimuli, internal communication and timing dependencies. The pattern allows to differentiate between functional data and adaptive control data. Furthermore, we enable the modular verification of functional and adaptation behaviour using the notion of process refinement in Timed CSP. The verification of refinements and crucial properties is automated using industrial-strength proof tools.

Keywords: Adaptive Systems · Modelling · Verification · Timed CSP

1 Introduction

Modern adaptive systems are distributed among different network nodes. One of the advantages of (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. This, however, makes these systems very complex to design and analyse.

In this paper, we present a generic design pattern for distributed adaptive real-time systems. Its aim is threefold. First, it describes an architecture, which helps formally designing adaptive systems. Second, it enables a strict separation of functional and adaptation behaviour. Third, due to this separation, it allows for modular refinement of adaptive systems and thereby facilitates formal verification of possibly crucial properties.

As in our previous work [3], we enable the refinement-based verification of adaptation and functional behaviour. However, in this work we focus on a strict distinction between functional data and control data following [4]. This enables the separate verification of functional and adaptive properties. A functional component manipulates its functional data but may be controlled by possibly dynamic control data that can only be changed by some corresponding adaptation component. The adaptation component gathers information of the

© Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2015
P.C. Vinh et al. (Eds.): ICTCC 2014, LNICST 144, pp. 3–12, 2015.
DOI: 10.1007/978-3-319-15392-6_1

functional component, which it uses for an analysis concerning whether or not adaptation is necessary. Then, a plan is created that results in new control data, which is finally set in the functional component or sent to another distributed adaptive component. The separation of functional and control data allows for the clear separation of functional and adaptation components, which allows for modular verification. We show how this idea can be modelled and verified with Timed CSP in a modular and stepwise manner using automatic tool support.

The rest of this paper is structured as follows. In Section 2, we briefly introduce the process calculus Timed CSP and then discuss related work in Section 3. In Section 4, we introduce our timed adaptive specification pattern and discuss its refinement and verification capabilities. We illustrate the benefits of our approach using an example in Section 5. Finally, we conclude the paper in Section 6 and give pointers to future work.

2 Timed CSP

Timed CSP is a timed extension of the CSP (Communicating Sequential Processes) process calculus [9]. It enables the description and the compositional refinement-based verification of possibly infinite-state real-time systems. To this end, process operators like *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. 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 [5], which is the standard tool for CSP, tock-CSP is supported via timed operators and the `prioritise` operator with the internal τ event and other events can be given priority over `tock`. This is necessary to inherit the notion of refinement and its compositional features from Timed CSP. Refinement is usually considered in the semantical traces or failures model. This means, for example, that the refinement $P \sqsubseteq_T Q$ expresses that $traces(Q) \subseteq traces(P)$ where $traces(_)$ denotes all finite traces of a process.

3 Related Work

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

The work in [1] provides a development approach for adaptive embedded systems starting with model-based designs in which adaptation behaviour is strictly separated from functional behaviour. Verification is based on transition systems which are connected by input and output channels. Our approach aims to support development processes for adaptive systems with the powerful notion of CSP refinement and the mature proof tools for automatic refinement checking.

In [7], CSP is used to model self-adaptive applications where nodes in a network learn from the behaviour of other nodes. Behavioural rules of nodes are described by CSP processes, which are communicated between the nodes and used to adapt the individual behaviour. Our work focusses on modelling entire adaptive systems and verifying properties of the modelled systems.

Timed automata are used in [6] for modelling and verifying a decentralised adaptive system. Verification of crucial properties is done using the Uppaal model checker. In contrast, we focus on the stepwise development of and modular verification of distributed timed adaptive systems.

In [8], 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 have presented an approach for the specification and verification of untimed distributed adaptive systems in CSP. A main goal of this work was the separation of functional behaviour from adaptation behaviour. The application of this framework in [10] 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 their respective parts of the system variables are separated, it remains rather unclear how the interface between them can be modelled in a systematic manner. This drawback is addressed in this paper. Furthermore, we introduce mechanisms to specify and verify timed adaptive behaviour.

4 Timed Adaptive System Pattern

In this section, we introduce an abstract pattern for timed adaptive systems. It describes a general structure of timed adaptive systems, which is amenable to modular refinement-based verification. In Figure 1, the overall architecture is illustrated. We consider adaptive systems that consist of a network of adaptive components (AC(i)) that communicate using channels. Communication channels are categorised, depending on their origin, as either functional channels (FE) or adaptation channels (EA). A single component can perform some computation (also depicted by FE) or adapt its internal behaviour (IA) due to the violation of some (local) invariant. 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, functional events of a certain class have not been communicated. When some internal adaptation takes place, other components can be triggered to adapt their behaviour accordingly using EA events as introduced above. The environment interacts with the adaptive system using functional events only. As it might be necessary to restrict the behaviour of the environment, it can be constrained using the process ENV.

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. Below, we sketch how the

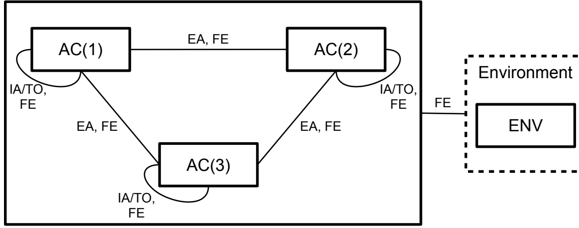

Fig. 1. Architecture of our Adaptive Pattern

described pattern can be formally defined on an abstract level in Timed CSP and how it can be refined in a stepwise way so that the verification of properties can be performed in a modular manner. The primary focus of the models lies on the separation of functional behaviour and adaptation behaviour. The refinement calculus of Timed CSP allows us to verify both of these aspects separately while leaving out concrete details of the respective other part. In addition to functional and adaptive behaviour, also timing behaviour can be specified and verified. To this end, we use the real-time capabilities of Timed CSP and FDR3.

4.1 Abstract Model

The overall 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. As the adaptation component has no direct access on the functional or control data, it has to explicitly fetch 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, thereby implementing IBM's MAPE (monitor, analyse, plan, execute) approach. This is captured in the `CHECKADAPT` and `ADAPT` processes described below. The adaptation component can also be triggered by some external adaptation event 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)
```

To check whether adaptation is necessary, the current (necessary) functional data and control data is fetched from the functional component. According to

local violations of the invariant, actual adaptation of control data takes place. On this level of abstraction, 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 in which adaptation takes `i` 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. Notification is realised in `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 and the control data can be set by the adaptation component. On this abstract level, we abstract away state information using constructions based on internal choices. The functional component can also communicate with other functional components or manipulate its functional data.

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

The abstract components have a far smaller state space than the refined components that we introduce in the following subsection. Only by this, the verification on the abstract level is possible 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, state information of the components is not present. A refined model needs to make clear when the actions actually take place. To do this in the context of CSP, non-determinism is usually reduced by replacing

internal choices (`|~|`) with guarded deterministic choices (`[]`). For the adaptation component, 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 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, e.g., guards (`gf(...)`).

```
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 -> FC'(i,h(d,x),cd))
```

In the next section, we explain the refinement and verification process in the context of the presented adaptive system pattern. This makes it also clearer why it is beneficial for a designer to provide models on different abstraction levels.

4.3 Proving Refinement

The aim of the described pattern is to facilitate the modular refinement and verification of adaptive real-time systems. By separating functional from adaptive behaviour, we are able to verify the respective properties separately.

The most abstract system model leaves out most of the details concerning adaptation and functional behaviour. When adaptation takes place, it has no direct influence on the functional behaviour of the components. Thus, the most abstract model is suited to verify properties, which focus neither on the adaptation behaviour nor the functional behaviour. By introducing detailed adaptation or functional behaviour, we refine the abstract model to a refined model that fulfils more required properties w.r.t. adaptation behaviour or w.r.t. functional behaviour due to the preservation of properties. The key point is that for many properties only the functional behaviour or the adaptation behaviour needs to be refined, not necessarily both at the same time.

A refinement-based verification approach has two major advantages. First, we can verify functional correctness and adaptation correctness separately. On the most abstract level, we have a system that is composed of a functional component FC and an adaptation component AC. Both of these abstract components leave out most of the details. By refining the functional component to FC' and the

adaptive component to AC', we can verify functional properties and adaptation properties while leaving out details of the respective component, which is not of interest for the respective property. Formally, we have $FC \otimes AC \sqsubseteq_{FD} FC' \otimes AC$ and $FC \otimes AC \sqsubseteq_{FD} FC \otimes AC'$. Furthermore, we have that $FC' \otimes AC \sqsubseteq_{FD} FC' \otimes AC'$ and $FC \otimes AC' \sqsubseteq_{FD} FC' \otimes AC'$. This means that all properties that are valid on the partly refined models $FC' \otimes AC$ and $FC \otimes AC'$ remain valid in the refined model $FC' \otimes AC'$. The second advantage is related to the environment model. In CSP, a model is more abstract than another when it contains fewer constraints. This means that a refined system has fewer behaviours than an abstract one. Ideally, we would like to verify an adaptive system with a most abstract or most unconstrained environment. However, this is almost never possible especially in the context of adaptive systems. Refinement allows us to include necessary constraints to the environment to prove the overall system correct.

5 Example

In this section, we present a simple adaptive system with which we 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 possibly should adapt to the new situation by changing the dim intensity. Furthermore, the dimmer can be adjusted manually, which represents the actual functional behaviour of the dimmer. On the abstract level, we omit details concerning the state information in the components. This means that all choices, which should depend on the state information are realised by internal choices.

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, the dimmer is possibly notified using the `newIntensity` event. On this abstraction 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 at most once a second the system is interacted with. This is a severe restriction but eases presentation. 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 = ((AdaptiveComponent1_00 [|{|newIntensity|}|]
              AdaptiveComponent2_00)
                [|{|light, higher, lower|}|]
  ENV) \{| newIntensity, getCurrent, getIntensity |}
```

We have modelled three safety properties as CSP processes. Thus, trace refinement is sufficient to express that a system fulfils them. Due to the lack of space, we omit their CSP definitions here. 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 there are only small jumps in the dimmer. The dim value before and after setting it can differ by two at most.

These properties are not valid in the abstract model presented above. We first need to refine the model to be able to show them. All three properties are concerned with the adaptation behaviour of the two components. So we need to 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 adaptation components above are updated with these refined parts accordingly (taking 0 as the initial value for `x`). The corresponding new system description `System_abs2` is sufficiently refined to show the second property using FDR3. Note that we need to prioritise internal events over `tock` and have to specify that the `setGoal` and `obs` events are urgent but visible.

```
assert P2 [T= prio(System_abs2,<{|setGoal,obs|},{tock}>)
```

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

```
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)
```

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

```
assert P1/P3 [T= prio(System_abs3,<{|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. Although being a relatively simple example, we are confident that we benefit from applying our approach to more complex systems as described in the next section.

6 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. It clarifies how

functional and control data is processed and communicated within the individual components of a distributed adaptive system. Adaptation is achieved in a decentralised fashion. Moreover, we have demonstrated how timing dependencies of adaptation behaviours can be modelled and analysed. 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 using automated verification methods.

In future work, we plan to apply our approach to an adaptive multicore system, which was previously only incompletely verified [10], because of limited scalability due to not 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.

References

1. Adler, R., Schaefer, I., Schuele, T., Vecchié, E.: From model-based design to formal verification of adaptive embedded systems. In: Butler, M., Hinchey, M.G., Larrondo-Petrie, M.M. (eds.) ICFEM 2007. LNCS, vol. 4789, pp. 76–95. Springer, Heidelberg (2007)
2. Allen, R.B., Douence, R., Garlan, D.: Specifying and analyzing dynamic software architectures. In: Astesiano, E. (ed.) ETAPS 1998 and FASE 1998. LNCS, vol. 1382, pp. 21–37. Springer, Heidelberg (1998)
3. Bartels, B., Kleine, M.: A CSP-based framework for the specification, verification and implemenation of adaptive systems. In: 6th Int. Symp. on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2011). ACM (2011)
4. Bruni, R., Corradini, A., Gadducci, F., Lluch Lafuente, A., Vandin, A.: A conceptual framework for adaptation. In: de Lara, J., Zisman, A. (eds.) Fundamental Approaches to Software Engineering. LNCS, vol. 7212, pp. 240–254. Springer, Heidelberg (2012)
5. Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A.W.: FDR3 — A modern refinement checker for CSP. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014 (ETAPS). LNCS, vol. 8413, pp. 187–201. Springer, Heidelberg (2014)
6. Iftikhar, M.U., Weyns, D.: A case study on formal verification of self-adaptive behaviors in a decentralized system. In: Kokash, N., Ravara, A. (eds.) FOCLASA. EPTCS, vol. 91, pp. 45–62 (2012)
7. Jaskó, S., Simon, G., Tarnay, K., Dulai, T., Muhi, D.: CSP-based modelling for self-adaptive applications. Infocommunications Journal LVIV (2009)
8. Luckey, M., Engels, G.: High-quality specification of self-adaptive software systems. In: 8th Int. Symp. on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2013). ACM (2013)
9. Schneider, S.: Concurrent and Real Time Systems: The CSP Approach. John Wiley & Sons Inc., New York (1999)
10. Schwarze, M.: Modeling and verification of adaptive systems using Timed CSP. Master thesis, Technische Universität Berlin (2013)