A Refinement Checking Technique for Contract-Based Architecture Designs ⋆

Raphael Weber[1], Tayfun Gezgin[1], and Maurice Girod[2]

[1] OFFIS, Escherweg 2, 26121 Oldenburg, Germany,
{raphael.weber,tayfun.gezgin}@offis.de
[2] Airbus Operations GmbH, Kreetslag 10, 21111 Hamburg, Germany,
maurice.girod@airbus.com

Abstract. During the development of software intensive systems, typically several models of this system are designed. These various models represent the system structured by different concerns, e. g. abstraction. While these approaches help to cope with complexity, the need of relating the models to one another arises. A major task is to keep model specifications consistent and traceable through special relations. The relation of interest for this work is the refinement relation between abstraction levels. In this work we describe a technique to check the validity of these refinement relations with respect to formal behavior/interface specifications of design items. For evaluation, we apply our refinement technique to an industrial example modeled with the contract-based methodology from our previous work.

1 Introduction

In recent years, the design process of systems in domains like automotive, automation technology, avionics, or consumer electronics has become a more and more complex task: The increasing number of functions which are realized by software, inter-dependencies of software tasks, and the integration of existing sub-systems lead to highly complex software intensive systems. This complexity in conjunction with the increasing demand on a short time-to-market and on quality aspects make it difficult for engineers to develop such systems. In order to cope with this difficulty we proposed a new meta-model along with a methodology in [2] to support the system architect.

The proposed common systems meta-model (CSM) is based on the meta-model of *Heterogeneous Rich Components* (HRC) [6], which provides the concept of contract-based specifications. The term “rich” alludes to the key ingredient of HRCs to provide rigorous interface specifications for multiple *aspects*, encompassing both functional and extra-functional (e. g. safety and real-time) characteristics of components. To structure the design space and enable different development approaches (e. g. top-down, bottom-up, . . .), we described the

⋆ This paper was partly funded by the German Federal Ministry of Education and Research (BMBF) through the project “Software Platform Embedded Systems (SPES 2020)”, grant no. 01IS08045.

concept of *Abstraction Levels* and *Perspectives* along with their CSM representation. For the designer to navigate and model relations between abstraction levels and perspectives, we briefly introduced the concept of a *Mapping* relation (called *Realization* between abstraction levels and *Allocation* between perspectives). Analyzing these mappings was not discussed in detail in our previous work and is subject of this work.

For evaluation, an industrial example was proposed by Airbus: the Air-Conditioning-System. We evaluate a certain portion of that example through refinement check. This check finds out whether a component of a specific design can be replaced by a component of a different design, i. e. it finds out whether the different design component can be virtually integrated into the environment of the other component. The refinement check is done via model checking an Uppaal [9] timed automata representation of the formal requirements of the model.

By formal requirements we mean the specification of requirements via the pattern-based requirements specification language (RSL). The RSL is developed and evaluated in the European CESAR project. While this requirement specification technique allows for easy transformation from predefined patterns to automata it may not be the most intuitivly usable one. Yet, there are a variety of methods to help to derive natural language requirements down to pattern based requirement specifications (contracts). However, that subject is not part of this paper, for more information about the RSL and requirement specification methods in general refer to [4].

Related Work In current literature one can find many works that deal with refinement checks via timed automata. One similar work is [15] in which the authors use Uppaal timed automata (among others) to verify the correctnes of their requirements. However, timed automata are not used to formally check the refinement of these requirements. The theoretical foundation and practical application of the contract-based specification method were elaborated in [5, 3]. One could consider our work as a follow-up work of [5]. In [13] the authors propose a basic calculus for adding and removing channels and components in a dataflow architecture. The calculus formally allows for refinement checking between two system architectures. While this approach covers adding and removing entities it does not include the modification of artifacts. Furthermore, there is no relation to meta-modelling concepts or to the (then emerging) UML standard. There is also no practical example on which the calculus was applied and evaluated. There is, however, an introducing example to motivate the approach.

The remaining parts of this paper are organized as follows. In the next section we give a short introduction on the modeling concepts of the CSM. In Section 3 we describe the refinement check for the case in which one contract is refined by exactly another contract and give the construction principle of the automaton networks. Section 4 illustrates the air-conditioning-system example by Airbus and explains how we applied the refinement check to the example. In Section 5 we discuss the extension of the refinement check where the specification of a component may consist of a set of contracts. In the last section we will sum up our findings and draw a conclusion.

2 Modeling Concepts

In our previous work [2] on a new common systems meta-model (CSM), Heterogeneous Rich Components (HRCs), which originate from the European SPEEDS project [17], represent the major modelling artifact. In addition to HRCs a new methodology to traverse along the design space is also proposed in [2]. This section gives a short introduction to the concepts of HRC and the design methodology of the CSM.

2.1 Heterogeneous Rich Components

The CSM provides basic constructs needed to model systems like components with ports and connections (bindings) between them. We refer to these components as Heterogeneous Rich Components (HRCs). HRCs rely on the basic concepts of SysML blocks [12]. The dynamics of an HRC can be specified by behavior, e. g. an external behavior model, or even source code. Furthermore, requirements (or contracts) refer to a *required* behavior whereas the *actual* behavior is specified as stated above. The idea of contracts is inspired by Bertrand Meyer's programming language Eiffel and its *design by contract* paradigm [10].

In HRC, contracts are a pair consisting of an assumption and a guarantee, both of which are specified by some text. Here we assume that assumptions and guarantees are specified by a pattern based formalism called requirement specification language. An assumption specifies how the context of the component, i. e. the environment from the point of view of the component, should behave. Only if the assumption is fulfilled, the component will behave as guaranteed. This enables the verification of virtual system-integration (integrate a more detailed component or a subcomponent in a more abstract environment) at an early stage in a design flow, even when there is no implementation yet. Thus, the system decomposition can be verified with respect to contracts. Details about the semantics of HRC are given in Section 3.1. Note that in this work we consider only the HRC semantics where a connection between two ports describes their equality. A deeper insight into HRCs can also be found in [8, 16].

2.2 Structuring the development process

When developing an embedded system, an architecture is regarded in different *Perspectives* at several *Abstraction Levels* during the design process as mentioned in Section 1. On each abstraction level the product architecture is regarded in different perspectives. As an additional concept for separation of concerns, models in each perspective reflect different aspects. For example, an aspect "Safety" might be regarded in every perspective but an aspect "Realtime" is not regarded in a geometric perspective and aspect "Cost" is not regarded when considering operational use cases.

2.3 Realization and Allocation

In order to keep the models in different perspectives and abstraction levels consistent (keep traceability between development steps) we defined a so called *Realization-* and *Allocation-Link*. The basic idea behind both concepts is to relate the observable behavior of components exposed at its ports.

Realizations are relationships between ports of components on different abstraction levels. Intuitively a realization-link states, that a component (e. g. *f1*) has somehow been refined and is now more concrete in terms of its interface and/or behavior (e. g. *f1'*). The refinement cannot always be captured by a pure decomposition approach. Thus, we define the realization of a component by introducing a state-machine that translates the behavior of the refined component *f1'* into according events observable at a port of component *f1*.

Allocations are relationships between ports of components in different perspectives. Intuitively an allocation-link states that the logical behavior of a component (e. g. *f4*) is part of the behavior of a resource (e. g. *r2*), to which it has been allocated. Here, we consider the same link-semantics as for the realization. For more details, refer to [2]. In the next section we will deal with the automatic verification of allocation and realize links in more detail.

3 Refinement Check

In the previous section we introduced our methodological concepts of the CSM and the HRCs. This section describes how the concepts of realize and allocate links can be automatically validated. For this, we first give an in-depth description of the semantics of HRCs.

The concepts of realize and allocate links between perspectives and abstraction levels are very similar. Both links define a refinement relation between components with respect to their specification: The refined components have to respect the requirements specified for their abstract counterparts. We say, a specification C' refines another specification C if and only if the behavior specified by C' is a subset of the behavior specified by C. In the following we will formalize the refinement relation with respect to contracts and give a technique in order to automatically check such a relation. In this work we will only consider 1-to-1 mappings, i. e. where a component is refined by exactly another component. Mappings, where a set of components is related to another set of refined components is subject of future work. Furthermore, we will only consider specifications which consist of exactly one contract. The generalization to a set of contracts is subject of Section 5.

3.1 Semantics of HRC

The specification of HRCs is given in terms of contracts over their interaction points capturing the required dynamics of a component. This means that for specifically assumed environment conditions the component shall guarantee a specific behavior exposed at its ports. In the following we assume without loss of generality that each port contains exactly one interaction point, so we can refer to that interaction point when talking about a port.

A contract is a tuple (A, G), where A defines the assumption and G the guarantee as introduced in Section 2. The semantics of a contract is defined as

$$[\![C]\!] := \; [\![A]\!]^{Cmpl} \; \cup \; [\![G]\!], \tag{1}$$

where $(X)^{Cmpl}$ defines the complement of a set X in some universe $\mathcal{U}$ and $[\![X]\!]$ is defined as the semantic interpretation of X.

The semantics of A and G is given in terms of sets of timed traces. A trace over a port set P is a sequence of port assignments. A port assignment $\mathcal{V}$ is a function $\mathcal{V}: \; P \rightarrow D$ assigning each port $p_i \in P$ to a value in its domain D_i. Further, a time sequence τ is a monotonically increasing sequence of real values, such that for each $t \in \mathbb{R}$ there exist some $i \geq 1$ such that $\tau_i > t$. A timed trace is a sequence (ρ, τ) where ρ is a sequence of port assignements and τ a time sequence. The set of all timed traces over P is denoted by $Tr(P)$.

The specification S of a component is given in terms of a set of contracts, i. e. $[\![S]\!] := \bigcap_{i=1}^{n} [\![C_i]\!]$. An implementation I of a component satisfies its specification S, if $[\![I]\!] \subseteq [\![S]\!]$ holds. The refinement relation between two contracts C and C' is defined in a similar way. Note that the definition for n contracts is subject of Section 5.

$$C' \text{ refines } C\text{, if } [\![A]\!] \subseteq \alpha([\![A']\!]) \text{ and } \alpha([\![C']\!]) \subseteq [\![C]\!], \tag{2}$$

where $\alpha : Tr(P') \rightarrow Tr(P)$ is called the mapping function (represented through a state machine as mentioned in Subsection 2.3) relating concrete traces with abstract ones. This function is necessary as both contracts C and C' may talk about different ports. Here we only assume mapping functions which can be transformed to timed automata. Note that mapping functions are not generated automatically but rahter are given by the systems architect.

In the following subsection we will introduce a technique to check such a refinement relation.

3.2 Checking the Refinement Relation

We introduce our concept of verifying a refinement relation by specifications consisting of only one contract. In order to check the refinement relation between the contract of the abstract and the concrete component, we derive Uppaal timed automata[9] out of both contracts and do a reachability check[1]. As defined in Equation 2 the check consists of two parts, i. e. first checking the set inclusion of the assumptions and second checking the set inclusion of both contracts.

Checking Set Inclusion of Assumptions In order to check $[\![A]\!] \subseteq \alpha([\![A']\!])$ we derive a timed automaton out of A' serving as passive observer O. The transitions of O are annotated with receiving events (derived out of port names) and clock constraints in such a way, that the observer accepts the set of timed traces which are element of $[\![A']\!]$. For all traces which are not element of $[\![A']\!]$ the observer enters a bad state. Then we derive an automaton T_A out of A which serves as a trigger for O. The transitions of T_A are annotated with sending events and timing constraints, such that T_A produces all traces that are element of A.

At least we need the automaton Gl realizing the mapping function α which receives events from T_A and translates corresponding events to the observer. This automaton is assumed to be given by the system architect.

From all automata we build the automaton network $T_A \parallel Gl \parallel O$. If the trigger automaton now produces a sequence which is not element of A' — and therefore the subset inclusion property will be violated — the corresponding observer will enter a bad state. So we need to check $T_A \parallel Gl \parallel O$ against the following Uppaal query: $A\Box$ *not* $O.bad$.

Checking Set Inclusion of Contracts The second part consists of checking $\alpha([\![C']\!]) \subseteq [\![C]\!]$. This is done by deriving an automaton network $T_{C'}$ out of C' or more precisely one automaton for the assumption part and one for the guarantee part. Both automata serve as trigger for the observer network derived out of C. Analogously to the first part it holds that whenever one of the trigger automata does a step which is not defined in the contract of the abstract component, the corresponding observer will enter a bad state.

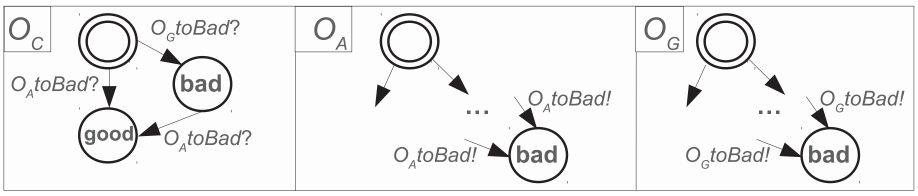


Fig. 1. Automaton network for Observer: O_G is the automaton for the guarantee, O_A for the assumption and O_C for the overall state of the contract.

The observer consists of three automata: one automaton for each assumption and guarantee and a third automaton tracing the state of the overall contract. This is illustrated in Figure 1: The observer obtained from the guarantee (O_G) and the assumption (O_A) send an event to the automaton O_C when entering their bad state. The automaton O_C states whether the contract is fulfilled. According to Equation 1 this is the case when either both the assumption and guarantee are fulfilled or the assumption does not hold. If the assumption does not hold, O_A sends an event O^A_{toBad} such that O_C directly switches to state `good`. If the guarantee was not fulfilled previously, then O_C switches to its good state in the case that the assumption is finally violated.

The automaton network $T_{C'} \parallel O_G \parallel O_A \parallel O_C$ is again checked against the Uppaal query: $A\Box$ *not* $O_C.bad$. If O_C enters its bad state as the corresponding guarantee is not fulfilled, we need to check whether finally its assumption automaton also enters its bad state, such that O_C switches to its state *good*. This is realized by checking the query $O_C.bad \dashrightarrow O_C.good$. The arrow is the *leads-to* operator of Uppaal and states for this case that whenever O_C enters its bad state it will finally enter its good state.

4 Case Study: Air-Conditioning-System

In this section we will describe in text and figures how the air-conditioning-system was modeled. Note that this is only a cut-out on two abstraction levels described in detail in Subsections 4.2 and 4.3.

4.1 Preliminary Notes on the Example

The following two subsections contain details about a cut-out of an example model of the air-condintioning-system (ACS) we did with Airbus in the scope of the SPES2020 project. This cut-out deals only with the technical perspective on two different abstraction levels, called the "upper" and the "lower". In each technical perspective we first define our system context (the environment outside the system under development). The boundaries of this environment description may change between abstraction levels, as is the case in our example. Note that we renamed some entities of the model.

The *air-generation demand calculation system* which is part of the ACS was previously modeled in the logical perspective (not described here). It is subject of the upper level whereas the processor of the upper level is refined in the lower level. The contracts of this particular processor and its subcomponents are described and examined for a refinement check which is described in Subsection 4.4.

4.2 The Upper Abstraction Level

The air-generation demand calculation system is designed in a rather fine level of granularity. At this point in the design process, it is time to initially design the technical architecture. Our system context contains the air-generation demand calculation system with four inputs (selected air demand from two crew selections and two sensor values of the actual climate to be conditioned) and one output (the calculated air demand value).

The system under design (the air-generation demand calculation system) is decomposed into certain resource artifacts. The *Input_ Terminal* (a device resource, fetching data from certain memory addresses), the *Input_ Com* (a communication resource, among others, sending requested data over an Input bus to a computation resource), the *CPU1* (a computing resource on which the actual air-generation demand calculation task is executed), the *Out_ Com* (a communication resource which, among others, sends the result of the calculation to the Out_Terminal, to be distributed), and the *Out_ Terminal* (a device resource, writing data to certain memory addresses). The schedule, according to which the task is scheduled, is a static time table, just like an ARINC653-Module specification (see [7]). The system under design with its artifacts is displayed in Figure 2.

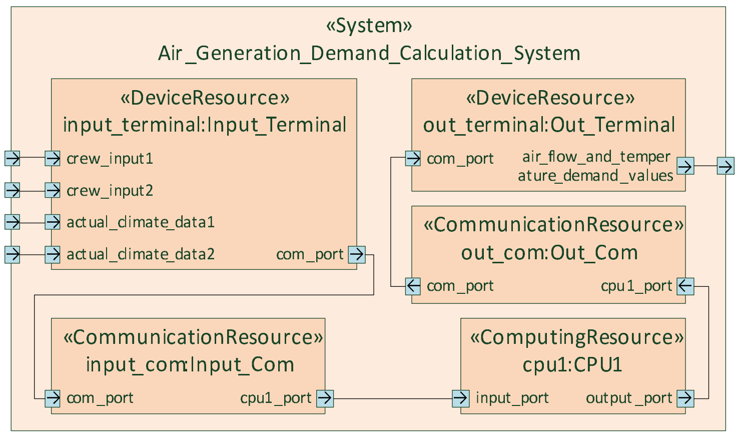


Fig. 2. The air-generation demand calculation system with resources.

The terminals and busses details are not described here. Instead we focus on the model of the *CPU1*. It contains one task (*demand_ calculation*), which performs the actual air flow and temperature calculation based on its inputs and writes it to its output port. Furthermore, the *CPU1* component contains a scheduler slot (which contains information, relevant for the scheduler) representing a certain amount of time within the schedule of *CPU1*. *CPU1* also contains the scheduler that has a certain scheduling policy. The ports between the slot and the task are SchedulerPorts, over which the slot "tells" the task when it is executing. Likewise, the ports between the scheduler and the slot enabling the scheduler to "tell" the slot when it executes. This scheme also makes hierarchical scheduling possible and allows other scheduling policies to be specified. The *CPU1* with its task, slot, scheduler, and a formal contract (denoted by "Contract_CPU_Performance") are displayed in Figure 3. The contract assumes that an event on *input_ port* occurs each 100ms and guarantees that the distance between the input event on *input_ port* and the output event on port *ouput_ port* is within 12ms and 15ms.

Note that *CPU1* has only one task receiving data and exporting data back to the bus. This indicates that there might be some bus interface functionality within the task. So maybe it is a good idea to decompose this one task a little further. The following section will describe the lower abstraction level (the component level), in which the technical perspective will be refined.

4.3 The Lower Abstraction Level

After designing the initial technical architecture on the upper level, we now proceed to model the technical architecture with more detail on the lower abstraction level. As mentioned above, we need to decompose the air-generation-demand-calculation-application task a bit further into the importer task, the

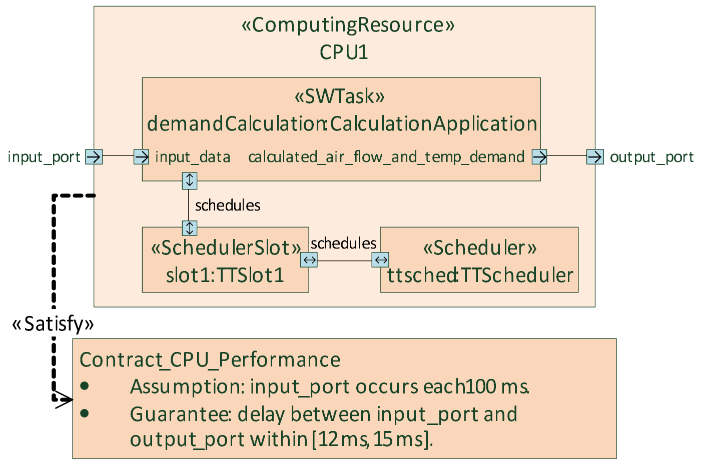

«ComputingResource»
CPU1
«SWTask»
demandCalculation:CalculationApplication
input_port
input_data
calculated_air_flow_and_temp_demand
output_port
schedules
«SchedulerSlot»
slot1:TTSlot1
schedules
«Scheduler»
ttsched:TTScheduler
«Satisfy»
Contract_CPU_Performance
• Assumption: input_port occurs each 100 ms.
• Guarantee: delay between input_port and output_port within [12 ms, 15 ms].

Fig. 3. The CPU1 component with one task.

actual calculation task, and the exporter task. This means, we will refine the computing resource *CPU1* from the upper level.

For the environment part we modeled the type with the two corresponding ports of the CPU. For now, there is no behavior environment model, so there is no further decomposition of the environment component. As for the CPU: It was decomposed into its tasks, slots and the scheduler. Figure 4 depicts the model.

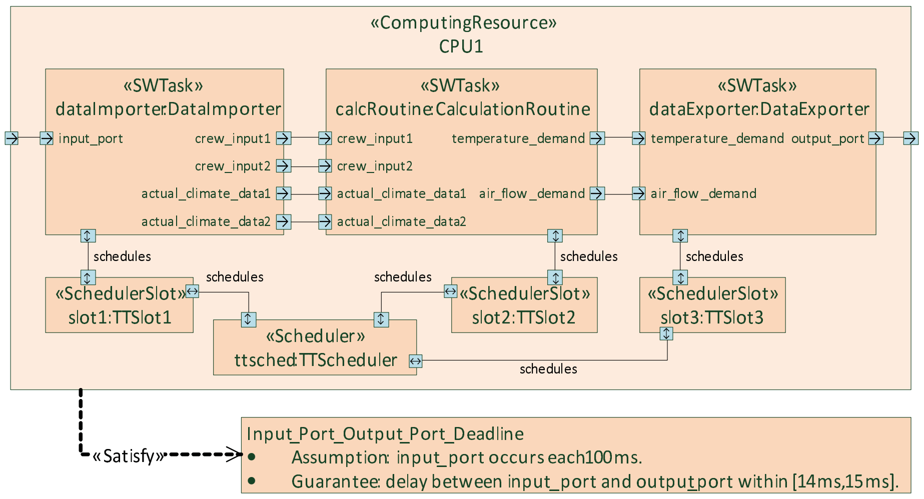


Fig. 4. The lower abstraction level CPU1 component with three tasks.

As depicted there are now three tasks in the computing resource: The *dataImporter*, *calculationRoutine*, and the *dataExporter*. The data importer has one input (the connection to the *Input_Com* communication resource like on the upper level). Likewise, the data exporter has the output port connecting to the *Out_Com* communication resource. The calculation routine has an input port for each incoming data and an output port for each outgoing data. It can only be activated if all four incoming data packets are available.

There is also a Contract (denoted by "Input_Port_Output_Port_Deadline") that the *CPU1* satsifies: The refined (from the upper level) deadline for the input and output port. Note that the assumption is unchanged, but the deadline interval has changed now (this often happens when traversing from a more abstract design to a more detailed design).

Having the two technical perspectives, one on the upper and one on the lower level, we now need to specify how they are mapped, i. e. how the upper level is refined into the lower level. Figure 5 shows how the task on the upper level is mapped to the tasks on the lower level. In this case it is rather simple, though the ports have different names their types are the same and it is a simple One-Port-To-One-Port mapping. Having said that, it is not obvious how the two contracts on both levels relate to each other.

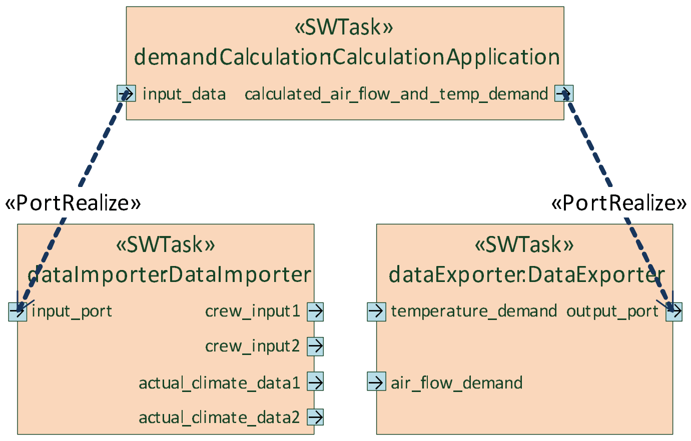


Fig. 5. The mapping between the task of the two abstraction levels.

Keep in mind that in the HRC semantics, a connection between two ports describes their equality. So, for delegation connections the contract's parts and ports are the same and for assembly connectors assumption and guarantee have to be checked against each other. In our case this means: The `input_port` of *CPU1* on the upper level is delegated to the task port `input_data` (see Figure 3) denoting their equality. The same is valid for the connection between `output_port` and the `calculated_air_flow_and_temp_demand`. Again, this also holds for the ports on the lower level (see Figure 4). So, with the contracts on the upper and lower level satisfied by the corresponding *CPU1*, parts of these contracts are also valid for the subcomponents of *CPU1*. It so happens that these subcomponents are mapped and thus induce the necessity of an realization test: Is the contract of the coarser component still valid if it is replaced by a finer component? Or in our example: Is the *CPU1* on the lower level a valid realization of the *CPU1* on the upper level, concerning their contracts and mappings? For this we apply the refinement check which formally checks if the answer to the above questions is Yes. The next subsection will describe in detail how this refinement check is performed for our example.

4.4 Realization Check between Abstraction Levels

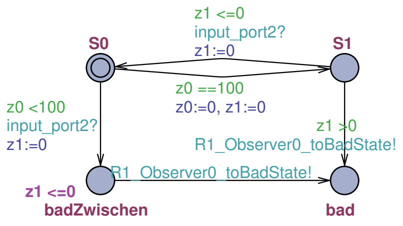
We will apply the technique checking a refinement relation introduced in Section 3.2. For this, we implemented a tool which derives relating contracts out of the system models, parses the RSL pattern with which the contracts are specified, generates a corresponding Uppaal timed automaton network and checks this against the properties illustrated in Section 3.2.

Consider again the contracts of *CPU1* in the upper and lower abstraction level introduced in the previous section. In general we have to check both conditions $[\![A_s]\!] \subseteq \alpha([\![A'_s]\!])$ and $\alpha([\![C']\!]) \subseteq [\![C]\!]$, but as in our example A_s and A'_s are equal, we can directly start checking the second condition. The observer automata resulting from the *CPU1* contract of the upper abstraction level are illustrated in Figure 6. It consists of three parts:

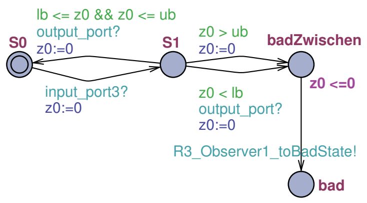
- The automaton O_A in Figure 6(a) results from the assumption of *CPU1* at the upper abstraction level, i. e. $A_s =$ *input_port* occurs each $100ms$. Each $100ms$ it enters its state S_1 and expects to receive event `input_port`. If the event is not received timely, it enters its bad state and sends an event `R1_Observer0_toBadState` to the automaton depicted in Figure 6(c).
- The automaton O_G in Figure 6(b) results from the guarantee part of the contract, i. e. $G =$ delay between *input_port* and *output_port* within $[12ms, 15ms]$. It enters its bad state if the delay between event *input_port* and *output_port* is less than $12ms$ or greater than $15ms$.
- The automaton O_C in Figure 6(c) gives the overall state of the contract. If O_A switches to its bad state, the contract is trivially fulfilled and O_C switches to the state `good`. If the O_G switches to its bad state, the O_A must finally switch to its bad state, because otherwise the contract would be falsified.

This observer network is triggered by the automaton network depicted in Figure 7 consisting of two automata:

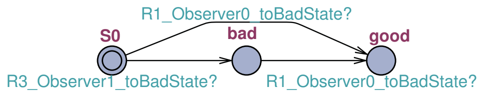
- The automaton in Figure 7(a) results from the assumption of the contract of *CPU1* at the lower abstraction level, i. e. $A_s =$ *input_port* occurs each $100ms$.
- The automaton in Figure 7(b) results from the guarantee part of the contract $G =$ delay between *input_port* and *output_port* within $[14ms, 15ms]$.



(a) Observer automaton for $A_s = input\_port$ occurs each $100ms$.

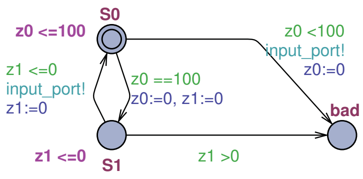


(b) Observer automaton for G = delay between *input_port* and *output_port* within $[12ms, 15ms]$.

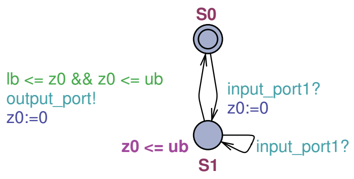


(c) Automata giving the state of the contract.

Fig. 6. Automata for the contract of the abstract component.



(a) Trigger automaton for $A_s = input\_port$ occurs each $100ms$.



(b) Trigger automaton for G = delay between *input_port* and *output_port* within $[14ms, 15ms]$.

Fig. 7. Automata for the contract of the concrete component.

This automaton network is checked against the property $A\Box$ not $O_C.bad$, where O_C is the observer automaton illustrated in Figure 6(c). This property states that the bad state of O_C is never reached. If the network fulfills this property, we have shown a valid refinement, which is the case in this example.

5 Considering Sets of Contracts

In this section we discuss the extension of the refinement check introduced in Section 3 in order to deal with specifications which may consist of a set of contracts. In this section we will omit technical details of the construction of automata networks like e. g. glue automata. Instead, we will focus on the general checking procedure and will give an extended formulation of the proof obligations introduced in Section 3. Further, we will discuss the construction for only a restricted subset of properties since (as we will see in the following subsection) that the generalization is not always applicable. For the assumption parts we will only consider activation patterns stating periodic triggering of a port. The guarantee parts will contain only delay patterns as demonstrated in the example of Section 4.4. More general cases are work in progress.

5.1 Extension of the Refinement Check

The specification of a component with more than one contract is obtained by conjugating them. Unfortunately, the conjunction of two or more contracts is not simply the conjunction of all assumptions and all guarantees. Rather there are various possibilities to formulate the conjunction of contracts. We will use the following:

$$C_1 \wedge ... \wedge C_n = \ (\mathcal{A},\ \mathcal{G}),\ \textbf{with}\ \mathcal{A} = \bigwedge_{i=1}^{n} A_i\ \vee\ \bigvee_{i=1}^{n}(A_i \wedge \neg G_i)\ \textbf{and}\ \mathcal{G} = \bigwedge_{i=1}^{n} G_i.$$

The first refinement property $[\![\mathcal{A}]\!] \subseteq [\![\mathcal{A}']\!]$ is checked in two steps (note that in the following we will omit the semantic evaluation brackets for the sake of readability) :

- $\bigwedge_{i=1}^{n} A_i \ \subseteq\ \mathcal{A}'$
- $\bigvee_{i=1}^{n}(A_i \wedge \neg G_i)\ \subseteq\ \mathcal{A}'$

We omit the necessity to construct a timed automaton for the trigger part which generates the union of both parts, by splitting this check into two parts. Note that timed regular languages are not closed under complementation. So this approach does not work in general. For the special case of delay patterns, the complement of the guarantee automata can be easily constructed.

The second refinement condition becomes in the general case $(\mathcal{A}',\ \mathcal{G}')\ \subseteq (\mathcal{A},\ \mathcal{G})$. In the first part of the check we have shown $\mathcal{A} \subseteq \mathcal{A}'$ which is equivalent to $\neg\mathcal{A} \supseteq \neg\mathcal{A}'$. With this we can simplify the left hand side of this term to $\mathcal{G}'$. For technical reasons it could further be necessary to add the assumptions to the guarantee. Consider for example the special case of activation and delay patterns: The assumption part is used to trigger the guarantee part. Without the assumptions an additional trigger structure for the guarantee part would be necessary.

In order to extend the left hand side of the above expression we use the following equivalence (zero set extension):

$$\mathcal{G}' = \ (\bigwedge_{i=1}^{n} A_i' \vee \neg(\bigwedge_{i=1}^{n} A_i')) \wedge \mathcal{G}'.$$

Because it holds that $\neg(\bigwedge_{i=1}^{n} A_i') \wedge \mathcal{G}' \subseteq \mathcal{A}$ we can omit this term. This leads us to the following proof obligation:

$$\bigwedge_{i=1}^{m}(A_i'G_i') \subseteq (\bigvee_{i=1}^{n} \neg A_i\ \wedge\ \bigwedge_{i=1}^{n}(\neg(A_i) \vee G_i) \vee\ \bigwedge_{i=1}^{n} G_i).$$

Again we have the problem of complementing a timed automaton. In the case of the automata A_p resulting from periodic activation patterns we can again find an automaton accepting all words which are not accepted by A_p.

5.2 Cyclic dependencies

If the specification of a component consists of a set of contracts, we could get cyclic dependencies if an output port is connected to an input port. For this we need a strong causality between events. In [11] this problem is prevented by a stepwise definition of contracts, i. e. all guarantees hold initially and if the assumptions hold up to the n^{th} step then the guarantees hold up to $n+1^{th}$ step. Another way to break such cyclic dependencies is to add delays to the data flow. The cases we considered utilized delay patterns.

6 Summary

In this paper we illustrated a technique to verify refinement relations for contract-based specifications for our previously proposed common systems meta-model (CSM). Our CSM allows to structure the design space by concepts like Abstaction Levels and Perspectives. It enables formal specifications via contracts allowing for each aspect to characterize the allowed design context of a component. In order to preserve traceability between model artifacts and to put those into refinement relations, the concept of Mapping was introduced.

In particular, an abstract component which is realized by a more concrete component has a refinement relation to that component. This relation has to respect contract specifications of both components. Thus, we formally defined the refinement relation and introduced a timed automaton based verification technique. For this, we derived a timed automaton network out of the assumption and guarantee parts of relating components and defined necessary properties. Whenever the network adheres to the properties, the refinement relation between the corresponding contracts holds. The resulting network was checked against the properties with the aid of the verification tool Uppaal. For evaluation, our technique was applied to an industrial case study from the avionics domain.

Currently, we are extending our approach of refinement checking in order to deal with more general n-to-m mappings, i. e. where one component is related to a set of more abstract or more detailed components. Especially cycles deserve more research: Interdependent contracts may lead to false conclusions. This problem is widely discussed in literature, e. g. in [11, 14].

In the future, we will analyze for which set of properties our approach is applicable. In this context we will analyze the effects which are occuring when we extend our approach to multi viewpoint analyses.

Further research will be conducted also in the field of deeper analysis methods for more complex mapping functions. In the future we also plan to integrate the refinement check described in this work with a broader evaluation of architecture alternatives in order to guide a developer through a design space exploration process.

References

1. L. Aceto, A. Burgueño, and K. Larsen. Model checking via reachability testing for timed automata. In B. Steffen, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *Lecture Notes in Computer Science*, pages 263–280. Springer Berlin / Heidelberg, 1998. 10.1007/BFb0054177.
2. A. Baumgart, P. Reinkemeier, A. Rettberg, I. Stierand, E. Thaden, and R. Weber. A model-based design methodology with contracts to enhance the development process of safety-critical systems. In *Proceedings of the 8th IFIP WG 10.2 international conference on Software technologies for embedded and ubiquitous systems*, SEUS'10, pages 59–70, Berlin, Heidelberg, 2010. Springer-Verlag.
3. A. Benveniste, J.-B. Raclet, B. Caillaud, D. Nickovic, R. Passerone, A. Sangiovanni-Vincentelli, T. Henzinger, and K. G. Larsen. Contracts for the design of embedded systems, Part II: Theory. Submitted for publication, 2011.
4. CESAR SP2 Partners. Definition and exemplification of requirements specification language and requirements meta model. CESAR_D_SP2_R2.2_M2_v1.000.pdf on http://www.cesarproject.eu/fileadmin/user_upload/, 2010.
5. W. Damm, H. Hungar, B. Josko, T. Peikenkamp, and I. Stierand. Using contract-based component specifications for virtual integration testing and architecture design. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1–6, march 2011.
6. W. Damm, A. Votintseva, A. Metzner, B. Josko, T. Peikenkamp, and E. Böde. Boosting re-use of embedded automotive applications through rich components. In *Foundations of Interface Technologies, FIT'05*, 2005.
7. A. R. INC. ARINC 653 - Avionics Application Software Standard Interface - Part 1 - Required Services. *Part of ARINC 600-Series Standards for Digital Aircraft & Flight Simulators*, March 2006.
8. B. Josko, Q. Ma, and A. Metzner. Designing Embedded Systems using Heterogeneous Rich Components. *Proceedings of the INCOSE'08*, 2008.
9. K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer STTT*, 1(1-2):134–152, 1997.
10. B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.
11. J. Misra and K. Chandy. Proofs of networks of processes. *Software Engineering, IEEE Transactions on*, SE-7(4):417 – 426, july 1981.
12. Object Management Group. *OMG Systems Modeling Language (OMG SysML ^{TM})*, November 2008. Version 1.1.
13. J. Philipps and B. Rumpe. Refinement of information flow architectures. In *Proceedings of the 1st International Conference on Formal Engineering Methods*, ICFEM '97, pages 203–, Washington, DC, USA, 1997. IEEE Computer Society.
14. A. Pnueli. *In transition from global to modular temporal reasoning about programs*, pages 123–144. Springer-Verlag New York, Inc., New York, NY, USA, 1985.
15. R. P. Pontes, M. Essado, P. C. Véras, A. M. Ambrósio, and E. Villani. Model-based refinement of requirement specification: A comparison of two v&v approaches. *ABCM Symposium Series in Mechatronics*, 4(IV.05):374–383, 2010.
16. Project SPEEDS: WP.2.1 Partners. SPEEDS Meta-model Behavioural Semantics — Complement do D.2.1.c. Technical report, The SPEEDS consortium, 2007.
17. The SPEEDS Consortium. SPEEDS Project. `http://www.speeds.eu.com`.