Reusing Dynamic Communication Protocols in Self-Adaptive Embedded Component Architectures

Christian Heinzemann
Software Engineering Group
Heinz Nixdorf Institute, University of Paderborn
Warburger Str. 100
D-33098 Paderborn, Germany
c.heinzemann@uni-paderborn.de

Stefan Henkler
OFFIS
Escherweg 2
D-26121 Oldenburg, Germany
stefan.henkler@offis.de

1. INTRODUCTION

In component based software engineering, the software under construction is assembled from different components. The partitioning of the software into components aims at reusing components in other systems in order to reduce development costs and time of future software systems. This approach is also applied for the specification of advanced mechatronic systems [9] that support self-* properties such as self-adaptation, self-management, or self-optimization and must fulfill real-time requirements. Mechatronic systems are mechanical systems combining elements from mechanical engineering, control engineering, electrical engineering, and software engineering. Often, such systems require a communication of one component instance with a varying number of other component instances, called communication partners, all being of the same type.

In [10, 16], it has been shown that reuse of components can be facilitated by separating protocols definitions from concrete components. A protocol definition specifies the required communication protocol independent of a concrete component implementation. The protocol is specified in terms of real-time statecharts [8] which are state machines with real-time extensions. A concrete implementation often has to refine this behavior, e.g. add internal computations or access internal variables, thereby introducing new internal states and/or transitions. In case of 1:n communication, i.e. one component instance communicates with a varying number of other component instances, the addition and removal of communication partners has to be considered when checking for a correct refinement. This problem is more difficult as the 1:1 communication case as additionally the creation and deletion of the protocols and the dependencies between the instances has to be considered. Since the abstract communication protocols are verified formally beforehand, the refinement must preserve these verified properties.

In the literature, two basic types of refinements are defined: simulation and bisimulation that exist for untimed systems as well as for real-time systems [4, 23]. These standard refinement definitions are based on automata and disregard varying numbers of communication partners which results in a varying number of real-time statecharts in our approach. Additionally, simulation is a very weak condition as it does not require the refined system to offer all allowed communications being required in the abstract protocol definition. Obviously, this is not sufficient for safe component reuse. In contrast, bisimulation is a very strong condition as it requires the refined system to perform exactly the same in exactly the same time as the abstract protocol definition. That does not allow to apply changes to the protocol thereby limiting the set of components complying to the abstract protocol.

In this paper, we introduce a refinement definition that relaxes the

Categories and Subject Descriptors
D.2.4 [Software/Program Verification]: Model Checking; D.2.13 [Reusable Software]: Reuse models; C.3 [Special-Purpose and Application-based Systems]: Real-time and embedded systems

General Terms
Algorithms, Verification

Keywords
strict conditions of a bisimulation by using information of our component model. In contrast to existing refinements, our refinement considers varying numbers of communication partners. For this, our refinement requires that the instantiation and deletion of communication links to these communication partners be performed in the correct time intervals. Along with the refinement definition, we provide a check method based on a common formalism for state-based real-time behavior supporting changes in the number of communication partners.

A concrete example for a complex self-adaptive system with the need to coordinate a varying number of component instances is the RailCab project. The vision of the RailCab project is a mechatronic rail system where autonomous vehicles, called RailCabs, apply new propulsion technologies to travel on the existing track systems. One particular problem is the convoy coordination of certain RailCabs [10]. RailCabs drive in a convoy in order to reduce energy consumption caused by air resistance and to achieve a higher system throughput. Such convoys are established on-demand and require small distances between the RailCabs. The required small distances imply real-time coordination between the speed control units of the RailCabs. Hence, this is safety-critical resulting in a number of constraints, that have to be addressed when designing the RailCabs’ control software. In addition, a complex coordination is required when the convoy consists of more than two RailCabs. Since RailCabs can join or leave a convoy during run-time, a flexible structure for the specification of the coordination is needed.

The contribution of this paper is a new refinement definition for in real-time communications with dependencies between the communication partners and a refinement check method. We relax the strict conditions of a timed bisimulation to ease the reuse of components and incorporate the change of communication links in the definition. The refinement definition can preserve safety and bounded liveness properties.

The paper is structured as follows: Firstly, we will explain the foundations of our component model, the MECHATRONIC UML, in Section 2. Thereafter, we present our refinement approach for MECHATRONIC UML in Section 3 followed by our evaluation results in Section 4. Related work is discussed in Section 5. We conclude with a summary and future work in Section 6.

2. MECHATRONIC UML

In this section, we briefly introduce MECHATRONIC UML that adapts concepts of the UML [20] to modeling mechatronic systems. In this paper, we will restrict ourselves to discrete software components. The concepts of the MECHATRONIC UML have been implemented as part of the Fujaba4Eclipse Real-Time Tool-Suite [21]. In the following subsections, we first describe parameterized coordination patterns and real-time statecharts which are used to model real-time communication protocols between components. Having defined those protocols, we will review the MECHATRONIC UML component meta-model in which the coordination patterns and real-time statecharts are used. Afterwards, we will introduce timed graph transformations specifying adaptations of our component architecture during run-time. Finally, we will present the timed story charts approach providing a common formalism for the refinement definition.

2.1 Parameterized Coordination Pattern

Parameterized coordination patterns [16, 15] are used to model communication protocols between communication partners. For example, in our RailCab use case, operating in a convoy requires one RailCab in the convoy to serve as a coordinator providing reference speeds and positions to the other convoy members. Therefore, this particular RailCab has to communicate with a set of other RailCabs changing during run-time. This implies that the coordination pattern has to be parameterized in the number of communication partners.

Figure 1: Parameterized Coordination Pattern

The communication partners are represented by roles. A coordination pattern consists of two roles and a channel connecting the roles as shown in Figure 1. A role can either be a simple role, as e.g. the member role, or a multi-role, as e.g. the coordinator role. A multi-role can be instantiated arbitrarily often while a simple role can only be instantiated once per pattern instance as shown in Figure 2. Instances of a multi-role are called sub-roles, sub-roles can be created and deleted by so-called adaptation statecharts at run-time to support varying numbers of communication partners.

The role behaviors are specified by real-time statecharts (cf. Section 2.2). Then, the roles of the coordination pattern can be assigned to the ports of a component. As a result, the port implements the behavior of the role and can refine this behavior as described in Section 3. This allows reusing proven coordination patterns for other components. In our example, the multi-role coordinator has a constraint [ordered]. This implies that each sub-role is assigned an instance parameter k and that an order exists on these parameters k.

Figure 2: Schematic Structure of a Multi-role

Parameterized Coordination Patterns can be verified for safety and bounded liveness properties [4] in order to assure correctness of the specified communication protocol. An example of a safety property is deadlock freedom as shown in Figure 1.

2.2 Parameterized Real-Time Statecharts

The behavior of each role is specified by a (parameterized) real-time statechart [8, 15] as depicted in Figure 1. The parameterization of the statechart refers to the parameters of the sub-roles of a multi-role. The general protocol being implemented by the pattern in the example is that the coordinator role sends an update containing a new reference speed and position to all member roles. The member roles answer with an acknowledgement.

In contrast to normal statemachines [20], real-time statecharts have clocks as well as constraints on these clocks. In our example,
the state complete has an invariant \( c_1 \leq 30 \) specifying that the state has to be left when the value of clock \( c_1 \) becomes greater than 30. Transitions can carry time guards restricting the execution of the transition to a certain time interval. \( \{c_1\} \) models clocks that reset a clock to 0. Real-time statecharts are semantically defined over timed automata [1]. In case of an ordered multi-role, the statechart can be parameterized using the instance parameter \( k \) of the multi-role.

Transitions of a real-time statechart can carry events, synchronizations, and side effects. Events model asynchronous communication between statecharts. A received event is placed in front of a ".", a sent event is placed behind it. Synchronizations model synchronous communication between statecharts running in different regions of the same composite state. They are defined analogously to Uppaal synchronizations (cf. [2]), i.e. they are marked with ">?" or ">!" for received or initiated synchronizations respectively. Side effects are methods executed when the transition fires. They are modeled like function calls with rounded brackets.

Real-time statecharts allow for parameterization of synchronizations as shown in Figure 1. There, the channel next is parameterized by the instance parameter \( k \). On instantiation of a sub-role, the parameter is assigned a concrete value with respect to the ordering of the sub-roles (cf. Section 2.1). Then, the parameter \( k + 1 \) of the synchronization \( next_{k+1} \) refers to the next sub-role. For instances \( k = 1 \) and \( k = 2 \) of the sub-role, e.g., it follows that the transitions complete \( \rightarrow \) idle of instance \( k = 1 \) and idle \( \rightarrow \) sendUpdate of \( k = 2 \) synchronize over the channel next. Thus, each sub-role triggers the next one.

The instantiation and deinstantiation of sub-roles of a multi-role is controlled by an adaptation statechart. Figure 3a) shows an adaption statechart for the coordinator multi-role. Whenever the state switches from noConvoy to addMember or from convoy to addMember, a new instance of the coordinator sub-role is created using the side effect createPort. The reconfiguration operation is then performed by a graph transformation as depicted in Figure 3b).

The adaptation statechart is also used to synchronize the sub-roles of the multi-role. The adaptation statechart triggers the first coordinator sub-role every 150ms to send an update. This is performed by the parameterized synchronization channel \( next_1 \) at the transition convoy \( \rightarrow \) sendUpdates. As described before, the parameter 1 refers to the instance parameter \( k \) of the first sub-role. The special synchronization \( next_{k+1} \) has no real corresponding sub-role because \( n \) gives the current number of sub-roles. In our example, we use this special synchronization for the last sub-role in the ordering, which has parameter \( k = n \), to synchronize with the adaptation statechart again to indicate that all updates have been sent.

2.3 Component Architecture

The MechatronicUML component model [10] is based on an own meta-model that is structurally similar to the UML 2.2 component model [20]. It consists of active components, i.e. every component executes a statechart defining its behavior. Our component model distinguishes between component types and component instances as we allow for multiple instantiation of components within a system. At run-time, component instances differ in the active states of their statemachines. Component types can be assembled hierarchically by embedding other component types as parts.

A component encapsulates its inner structure and behavior and allows interaction with other components only via its interfaces. In our component model, the interfaces are structured by ports. Therefore, a component type defines a set of named ports having at most one require and one provided interface. Since communication between ports is realized by asynchronous messages, each port has a queue for received messages to decouple sender and receiver. Figure 4 shows an example for a component type modeling a simplified RailCab with two component parts, Coordinator and DistanceControl, and four ports named member, coordinator, front, and rear.

The behavior of a port of a component is obtained by applying a role of a parameterized coordination pattern as introduced in Section 2.1 to the port. Then, the port behavior can refine the abstract protocol definition by, e.g., adding internal computations and accessing internal variable thereby adding/removing state and/or transitions. Components can only interact if they implement corresponding roles of the same coordination pattern as illustrated in the component instance situation depicted in Figure 5.
Timed Graph Transformations

In a parameterized real-time statechart, the side effects of transitions (cf. Section 2.2) are used to model internal computations as well as addition or removal of sub-roles of the respective multirole (cf. Section 2.2). Such side effects can be formally defined by using graph transformations [22]. Thus, parameterized coordination patterns are specified by a blend of real-time statecharts and graph transformations. In Section 2.5, we will introduce Timed Story Charts as a common formalism for real-time statecharts and graph transformations. We will use this common formalism for our refinement definition in Section 3.

Timed Story Charts are based on graph transformations, but need to incorporate the real-time extensions of our real-time statecharts. Therefore, we define Timed Graph Transformations [16, 13] as an extension of normal graph transformations in this section. As a result, we add the concept of real-valued clocks to graph transformation as well as time guards, invariants, and clock resets being defined for timed automata [1, 2] and real-time statecharts (cf. Section 2.2).

Time is measured by real-valued clocks like in real-time statecharts. In real-time statecharts, a clock applies to one statechart instance, i.e., each statechart instance for each sub-role has its own clock. Accordingly, in timed graph transformations a clock is valid for a certain subgraph of a graph. Such a subgraph may occur more than once in a graph and all occurrences might be created at different times. Therefore, the clock has to be added several times to the same graph (once for each subgraph it applies to). As all these instances of the clock can have different values, we use the term clock instance to refer to the clocks being added to the graphs. The clock serves as a type for the clock instances. Like in real-time statecharts, time passes at the same rate for all clock instances. By adding clock instances to the graph, we obtain a timed graph that can be defined as follows.

DEFINITION 1 (Timed Graph). Let $C$ be a set of clocks. A timed graph $G_t$ is a tuple $(G, C, I)$ where $G$ is an attributed, labeled graph and $C, I$ is a set of clock instances over the clocks in $C$. Each clock instance has a set of hasNode links to the nodes of the graph it applies to but no other edges.

We use UML Object Diagrams as a concrete representation of labeled attributed graphs and require compliance of the object diagram to a class diagram. The hasNode links of a clock instance to the other nodes of the graph induce the subgraph this clock instance applies to.

In contrast to normal graph transformations, we have to distinguish three kinds of rules. These are timed transformation rules, invariant rules, and clock instance rules.

Timed transformation rules correspond to the standard graph transformation rules. Thus, they change the object structure of the graph. Additionally, timed transformation rules may contain timed guards or resets on clock instances that are bound by the transformation. Figure 6 shows an example for a timed transformation rule specifying one transition of the timed story chart (cf. Section 2.5).

The rule is specified in a short-hand notation depicting left hand side and right hand side in one graph. Objects and links being created (or deleted) carry a stereotype <<++>> (or <<-->> respectively). It shows how the active state of a statechart can be changed. The rule specifies the timed guard $ci \leq 139$ for the clock instance $ci$ which is typed over the clock $c3$.

Invariant rules correspond to state invariants in real-time statecharts. They specify a left hand side but no right hand side. Additionally, they contain a time condition on a clock instance being bound by the left hand side. For each match that can be found for the left hand side, the time condition must be true for the match. If the invariant time is exceeded, the match must no longer be found in the graph.

Finally, clock instances rules can add clock instances to a subgraph of the timed graph. We refer to our technical report [13] for formal definitions of the rules.

The execution semantics of timed graph transformations is defined by the timed graph transitions system (TTS) [13]. The TTS can be obtained by a reachability analysis starting on the start graph of the timed graph transformation system [14]. It represents the whole reachable behavior and will be used as the basic data structure on which the refinement check (cf. Section 3.4) operates.

DEFINITION 2 (Timed Graph Transition System (TTS)). The Timed Graph Transition System (TTS) is a triple $(S, s_0, T)$ where $S$ represents the set of states of the TTS, $s_0 \in S$ is the initial state and $T$ represents the transitions. A state $s \in S$ is a tuple $s = (s_g, s_z)$ with $g \in G$ and $s$ a non-empty clock zone over the clock instances contained in $g$. In $s_0$, all clock instances are 0.

There exists a transition $t$ from $s_1$ to $s_2$, $s_1 \xrightarrow{t} s_2$, iff there exists a timed graph transformation rule such that $s_2$ is a successor state of $s_1$.

The states consist of a timed graph as defined in Definition 1 and a clock zone [1, 2]. A clock zone is a conjunction of constraints on the clock instances in the timed graph that defines the possible values these clocks may have. Intuitively, state $s_2$ is a successor state of $s_1$ w.r.t. a timed transformation rule if the rule transforms the timed graph of $s_1$ into the timed graph of $s_2$. A clock instance is non-empty iff the conjunction of clock constraints is satisfiable. The definition of the TTS is analogous to the definition of zone graphs [1, 2] for timed automata. Therefore, we refer to [14] or the literature on zone graphs for more detailed information on the computation of successor states.

States of the TTS are considered to be isomorphic if and only if the timed graphs are isomorphic and the clock zones are identical. The identification of isomorphic states leads to cycles in the TTS and is necessary for termination of the reachability analysis and the refinement check. Such cycles allow to represent infinite paths by a finite prefix and an infinite suffix represented by the cycle.
2.5 Timed Story Charts

Timed Story Charts are a common formalism for real-time statecharts and graph transformations. They are based on timed graph transformations using a specific meta-model for representing the real-time statechart. We will use Timed Story Charts in Section 3 for the refinement definition and for the refinement check.

The basic idea of timed story charts is to map the real-time statechart to an object diagram and a set of timed transformation and invariant rules (cf. Section 2.4). The object diagram encodes the states of the statechart as well as the active state. It is typed over the meta-model shown in Figure 7. Figure 8 shows an excerpt of an object diagram which is generated for the adaptation statechart shown in Figure 3 in which the coordinator is in state convoy.

![Timed Story Chart Meta Model](image)

Figure 7: Timed Story Chart Meta Model.

The class Statechart represents the statechart which contains a set of States and might be contained as a sub-statechart in a ComplexState. Considering parameterized statecharts (cf. Section 2.2), all instances have the same states. We exploit this fact by generating the objects for the states only once for all statechart instances. The concrete parameter \( k \) of the current instance is encoded as an attribute parameter in the ActiveState object pointing to the current state of instance \( k \). Parameterized synchronizations as used in Figure 1, e.g., are mapped to Synchronization objects. They contain the name of the synchronization as well as the parameter as attributes. Additionally, the class has links to the states in which the synchronization is available. Clock instances may have links to all elements within the statechart. These links are omitted in the figure for better readability.

The transitions of the statecharts are realized by timed transformation rules changing the active state and processing the events. Time invariants of the states are mapped to invariant rules (cf. Section 2.4). We refer to our technical report [13] for more details on the mapping.

3. REFINEMENT

Parameterized coordination patterns can be specified and verified independently of concrete components bound to its roles (cf. Section 2.1). This enables the reuse of coordination patterns for different components. But often, they cannot be reused without some modification, e.g., if values received as event parameters have to be stored inside the component or if multiple roles of different patterns have to be synchronized [10, 16]. In such cases, the role behavior has to be refined.

After such refinement, it is a priori not ensured that the resulting behavior specification is still compliant to the roles of the parameterized coordination pattern. In principle, there are two ways to ensure such compliance which we will introduce in Section 3.1. Afterwards, we will introduce a refinement definition based on timed story charts that allows to check such compliance in Section 3.2 and we will discuss its preserved properties in Section 3.3. Section 3.4 contains details about the refinement check procedure while Section 3.5 discusses its decidability.

3.1 Approach

The verification of properties for a parameterized coordination pattern ensures that the communication defined by that pattern adheres to these properties as shown in the upper part of Figure 9.

![Refinement Approach](image)

Figure 9: Refinement Approach

When applying such patterns to concrete components, the behavior of the roles may have to be adapted to these components as described before. In our example, the coordinator role statechart must send a new reference speed and position to the member role. The computation of such data is not part of the abstract member role statechart shown in Figure 1. Therefore, we need to refine the abstract role statechart to the refined coordinator role (cf. Figure 9). The refined coordination role is shown in Figure 10.
The statechart introduces a new state compData whose outgoing transition to sendUpdate executes a side effect computing the data to be sent. Additionally, the refined statechart merges the states awaitAck and complete into the state awaitAck thereby delaying the reception of the acknowledgement by one unit of time.

After such modifications, it is not guaranteed that the refined role is still compliant to the abstract pattern role which had been used for the verification of properties. Thus, it is not guaranteed that the communication between rc1 and rc2 of Figure 9 is safe. Now, there are two possibilities to verify this communication:

a) Verify both roles for correct refinement of the abstract role
b) Re-verify all properties for the refined protocol

If a role is a correct refinement of the abstract role (case a), a communication partner cannot distinguish it from the abstract role. Thus, all properties being verified for the parameterized coordination pattern will hold for the refined one as well. We say that the refinement preserves the properties of the abstract role. Case b) suggests a complete re-verification of all properties for the refined behavior specification which gives the same result as case a.

We believe that case a is much more efficient than case b for two reasons. Firstly, the verification of a complete pattern requires to build the product behavior of both roles including all sub-roles of the multi-role while the refinement check considers only one role at a time. Thus, the models to be handled by the refinement check are much smaller. Secondly, the re-verification has to be done for each property while the refinement checks the preservation of all properties in one step. Therefore, we will define a refinement for parameterized coordination patterns in the following subsection.

3.2 Definition

The refinement relation should preserve the verified properties of the pattern. The basic requirement is that a communication partner should not be able to distinguish a refinement from the abstract pattern role. As a result, the refinement has to be specified over the externally visible behavior in terms of sent and received events as well as the intervals between those events. It is also required that the refinement fulfills the complete externally visible behavior. The refinement can add internal behavior, but no external behavior, and it must fulfill the timing constraints of the abstract pattern role.

Therefore, the basis for our refinement definition are timed bisimulations (cf. [24, 23]). Strong and weak bisimulations exist. In a strong bisimulation, states originating from internal transitions (cf. [24, 23]). Strong and weak bisimulations exist.

In contrast to [23], we define our refinement based on timed story charts (see Section 2.5) rather than timed automata. In contrast to timed automata, timed story charts enable the checking of refinements of multi-roles including their state-based behavior and dynamic reconfiguration simultaneously.

This requires an additional structural refinement as defined in [12]. There, an abstraction function maps the types of the refined system to types of the abstract system and checks a subset relation $\subseteq$. The subset relation ensures that all necessary structural elements, e.g. ports or statechart instances, are present in both, abstract and refined behavior. This is important as a correct refinement of the state-based behavior is insufficient if the reconfiguration creating the corresponding structural elements is executed too late in the refined system.

Considering these issues, we can define our Relaxed Weak Timed Bisimulation as follows.

**Definition 4 (Relaxed Weak Timed Bisimulation).** Let $T_A = (S_A, s_0, A)$ and $T_R = (S_R, s_0, R)$ be timed graph transition systems where each state $s \in S$ is a tuple according to Definition 2. Let $\mu_A, \mu_R$ be the set of received and sent events and $\mu_{BO}$ a event. Furthermore, let $\Omega$ be a relation $\Omega \subseteq S_R \times S_A$ and $\abs$ an abstraction function mapping the types of the concrete system to types of the abstract system. Then, $\Omega$ is a Relaxed Weak Timed Bisimulation $T_R \simeq_{NTBS} T_A$ iff:

1. $(s_0, s)$ is a $\Omega$ and $\abs(g_{\Omega})$ and
2. $\forall (s, s') : s \xrightarrow{\delta} s' \Rightarrow \abs(g_{\Omega})$ and $\abs(s'')$ and $\abs(s''')$ and $\delta = \delta_1 + \delta_2$ for arbitrary $s''$, $s'''$.
3. $\forall (s, s') : s \xrightarrow{\delta} s' \Rightarrow (s'') \xrightarrow{\delta'} (s''') \xrightarrow{\delta''} (s')'$ for arbitrary $s''$, $s'''$.
4. $\forall (s, s') : s \xrightarrow{\delta} s' \Rightarrow (s'') \xrightarrow{\delta'} (s''') \xrightarrow{\delta''} (s')'$ for arbitrary $s''$, $s'''$.
5. ∀(s_R, s_A) ∈ Ω : s_R(δ_R → s_R| β_R → s_R) → s_R with δ_R = \sum δ_i

\textbf{Expression (A|B)^*} shall be interpreted as regular expressions and denotes sequences } x \in \{A, B\} \text{ of arbitrary length.}

Conditions 1 to 3 specify an untimed bisimulation on the events using the weak transition relation (cf. Definition 3). There, condition 1 requires the initial states of both TTS to be in the relation. Conditions 2 and 3 specify the two directions of the bisimulation. Conditions 4 and 5 specify the timing properties. Condition 4 states that the upper bound for sending an event in the refined system must be less or equal to the upper bound on the event in the abstract system for all clocks. Condition 5 states that the upper bound for an arbitrary event in the refined system must be greater or equal to the upper bound on the abstract bound for sending events in the abstract system. Together, conditions 4 and 5 state the desired property, that the upper bound for receiving an event can be relaxed whereas the upper bound for sending events must remain the same.

3.3 Preservation of properties

Our refinement definition specifies an untimed bisimulation on the events of both automata. Thus, all CTL* formulae referring to events can be preserved by the refinement [4]. The same holds for latest points in time at which events can be sent. Here, our refinement definition also defines a timed bisimulation on the set of sent events. Thus, all TCTL formulae can be preserved for them [23]. For received events, no timed formulae can be preserved as the latest point in time at which an event is received can be increased. Since we assume that the statechart has a buffer for incoming messages, this does not affect the other role as receiving events is always possible. In the refinement, we assume the queue to be an unbounded FIFO queue that holds messages until they are dequeued. In case of a different behavior, a real-time statechart representing the queue behavior has to be created and included in the refinement check.

In order to prevent an unbounded delay of received events, we require an alternating send/receive behavior, i.e., a statechart may not receive two events in a row without sending an event in between. In that case, receiving the event can only be delayed until the next event has to be sent. Since we consider hard real-time systems, this is no drawback as most communication patterns implement a watchdog behavior (events have to be acknowledged by the receiver)[6].

3.4 Refinement Check

The refinement check procedure introduced in this section verifies whether two roles, either two simple roles or two multi-roles, satisfy the refinement definition introduced in Section 3.2. The inputs for the refinement check are timed story charts of the two roles to be checked. The output is either an OK if the roles satisfy the refinement definition or a counter-example consisting of a path of the refined behavior that does not satisfy the refinement.

The refinement check consists of two steps. First, a reachability analysis as introduced in [14] is performed on both the abstract and the refined system, using the timed story chart. Then, the refinement check is performed on the resulting timed graph transition systems (TTS) (cf. Section 2.4).

The refinement check algorithm traverses the transition system for the refined system using a depth first approach. The algorithm investigates all possible paths in the TTS and identifies corresponding paths in the abstract system having the same event sequences.

For these paths, the conditions stated in Definition 4 are checked. A pair of corresponding states has to be identified for each event. The set of corresponding states builds the relation Ω. As it is not obvious which states will be in Ω, all possible combinations of states between two events have to be checked. Furthermore, it is possible that multiple candidate paths exist in the abstract system containing the same event sequence but have different timing constraints or structures. In this case, all possible candidate paths have to be investigated. If no path fulfilling the conditions of Definition 4 can be found, the check fails and thus the refinement is not correct.

Whenever an event is found along the path, a procedure called checkPath is invoked which checks the path between the last event and the current position. The input is a transition (n, n’) of the TTS of the refined system carrying an event. First, the path p_R from the last corresponding state pair in Ω to n is computed. If p_R does not contain an event, which will be the case if (n, n’) is the first transition on the refined path carrying an event, no corresponding states have to be found. Otherwise, an equivalent path p_A having the same events is searched in the abstract system starting at the corresponding state in Ω. Then, all possible combinations of states in p_R and p_A are checked for the timing requirements stated in Definition 4 and the structural refinement. If a pair of corresponding states has been found, they are added to Ω and the procedure returns. If no such path can be found, the refinement check fails.

In a TTS for our system, every path will end in a cycle (cf. Section 3.5). For cycles, there must be corresponding states between all events on the cycle. When a cycle is detected, the inputs from the initial state to the cycle have been checked for corresponding states that are now contained in Ω. Thus, the first run through the cycle has been investigated while checking the path. This is, however, not sufficient as there must be corresponding states between two consecutive events for all future runs through this cycle as well. In order to assure this, we have to analyze the cycle a second time. Examples for the possible cases that may occur, are illustrated in Figure 11. The grey states have been added to the relation Ω while checking the path.

**Figure 11: Considering Cycles in the TTS during Refinement Checking**

Situation (a) and (b) are OK because between each occurrence of an event along the cycle, there exists a state in Ω between two subsequent events. This is trivially true if the cycle does not contain an event at all. Situation (c) is not OK because there is no state in Ω along the cycle in the refined system. For situation (d), there is no state in Ω between events b and a for the abstract system.

After all traces of the refined system have been checked, the abstract system is traversed in order to check whether all paths of the abstract system have been covered, to ensure the bisimulation property. If this check is successful, the refinement is correct.
3.5 Decidability

In general, the verification of correct refinements based on trace inclusion is undecidable for timed systems [1] due to the possibility that the reachability computation may not terminate and paths may be infinite. In the case of mechatronic systems, we are exposed to hard real-time models typically running in a loop (cf. the examples in Section 2). In this case, every infinite path has a finite prefix only. The infinite suffix can be mapped to a loop in the timed graph transition system by identifying isomorphic states (cf. Section 2.4). Thus, the resulting timed graph transition system will be finite. Since finite problems are decidable, the verification of correct refinements is decidable for our class of problems.

4. EVALUATION

We have modeled the parameterized coordination pattern in Figure 1 with both, the abstract coordinator role as shown in Figure 1 and the refined coordinator role as shown in Figure 10 using the Fujaba4Eclipse Real-Time Tool-Suite [21]. Then, we transformed the patterns into a timed story chart (cf. Section 2.5) and used the reachability analysis introduced in [14] to check both patterns for deadlock freedom as this requires a complete exploration of the state space.

The verification of correct refinements as introduced in Section 3 considers only one role at a time. That requires a reachability analysis on the abstract and refined role behavior. For the evaluation, we considered the coordinator role including the adaptation statechart. Then, we used the resulting timed graph transition systems of these reachability analyses as inputs for the refinement check.

We performed all experiments for different maximum convoy sizes ranging from 2 to 11 which corresponds to 1 to 10 member RailCabs in the convoy. Table 1 summarizes the results that have been obtained for the four reachability analyses described above. We evaluated the number of objects in the timed graph transition system as well as the max. number of objects being contained in the graphs of the states. Both affect run-time and memory consumption of the reachability analysis. For a convoy size of 11 RailCabs, two of the reachability analyses resulted in an overflow error of an underlying library.

The refinement check as proposed in this paper is only sensible if it can be performed faster than a complete verification of the refined pattern. Therefore, Figures 14 and 15 compare the run-times and memory consumption of the check for deadlock freedom on the refined pattern as well as the run-time of a refinement check including the run-times for the reachability analyses on the abstract and refined coordinator port. The results show that the refinement check is much more efficient although it incorporates two reachability analyses and an additional algorithm.

As the refinement check mainly operates on the resulting timed graph transitions systems, the overall memory consumption of the refinement check is dominated by the memory consumption of the two reachability analyses. Compared to a full verification of the refined pattern, only about half the memory is needed for the refinement check (cf. Figure 15).
Table 1: Number of Reachable Graphs and Maximum Number of Objects per Graph

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>23</td>
<td>20</td>
<td>17</td>
<td>16</td>
<td>27</td>
<td>26</td>
<td>26</td>
<td>25</td>
</tr>
<tr>
<td>3</td>
<td>62</td>
<td>58</td>
<td>48</td>
<td>43</td>
<td>51</td>
<td>50</td>
<td>50</td>
<td>49</td>
</tr>
<tr>
<td>4</td>
<td>131</td>
<td>122</td>
<td>98</td>
<td>85</td>
<td>65</td>
<td>64</td>
<td>65</td>
<td>63</td>
</tr>
<tr>
<td>5</td>
<td>235</td>
<td>218</td>
<td>171</td>
<td>147</td>
<td>79</td>
<td>78</td>
<td>79</td>
<td>77</td>
</tr>
<tr>
<td>6</td>
<td>381</td>
<td>352</td>
<td>273</td>
<td>233</td>
<td>93</td>
<td>92</td>
<td>93</td>
<td>93</td>
</tr>
<tr>
<td>7</td>
<td>576</td>
<td>530</td>
<td>405</td>
<td>344</td>
<td>107</td>
<td>106</td>
<td>107</td>
<td>106</td>
</tr>
<tr>
<td>8</td>
<td>827</td>
<td>758</td>
<td>571</td>
<td>483</td>
<td>121</td>
<td>120</td>
<td>121</td>
<td>120</td>
</tr>
<tr>
<td>9</td>
<td>1141</td>
<td>1042</td>
<td>776</td>
<td>654</td>
<td>135</td>
<td>134</td>
<td>135</td>
<td>134</td>
</tr>
<tr>
<td>10</td>
<td>1525</td>
<td>1388</td>
<td>1025</td>
<td>861</td>
<td>149</td>
<td>148</td>
<td>149</td>
<td>148</td>
</tr>
<tr>
<td>11</td>
<td>error</td>
<td>error</td>
<td>1323</td>
<td>1108</td>
<td>error</td>
<td>error</td>
<td>error</td>
<td>error</td>
</tr>
</tbody>
</table>

In conclusion, our evaluation results clearly indicate that a refinement check is more efficient due to the smaller graphs being checked and the lower number of reachable graphs.

5. RELATED WORK

Our refinement check relates to several areas of research. Firstly, the refinement definition is applicable to self-adaptive embedded software changing its structure during run-time. Thus, our work relates to other techniques for modeling self-adaptive systems. Secondly, the MECHATRONIC UML uses a component architecture to model the system (cf. Section 2) and therefore, this work relates to other software component frameworks. Finally, our refinement check relates to other refinement definitions from the areas of graph transformations and timed automata as we combined ideas from both fields in our refinement definition (cf. Section 3). In the remainder of this section, we will discuss related work respectively.

In [3] an overview of modeling approaches for self-adaptive systems is presented. The approaches are intended to model adaptive behavior for non-real-time systems and fall mainly in three categories, namely: process algebra based, graph transformation based, and logic based. Although refinement definitions for some of the process algebra based approaches like LEDA exist, these are not directly applicable to our approach. Furthermore, the approaches do not consider real-time properties as they were not intended to. [25] describes a general procedure for modeling self-adaptive systems proposing a separation of adaptation behavior and behavior executed in the possible system configurations. We have used some of these ideas in our approach by encapsulating the adaptation behavior in graph transformation rules and modeling the remaining behavior in statecharts. The possible points in time where adaptation is allowed are determined by the calls to the reconfigurations thereby restricting the approach in [25].

The surveys [19], [5], and [17] review different component models. In [19] and [5] general purpose component models like e.g. CORBA, EJB, or SOFA are reviewed as well as component models for the development of embedded real-time systems like e.g. Progress, Robocup, or SOFA HI. The survey [17] focuses on component models for the development of embedded real-time systems, only. General purpose component models (like CORBA, EJB, ...), which are typically intended to model non-real-time systems, do not fit in the considered class of systems as real-time aspects are not considered.

In the area of component models for embedded real-time systems, Robocup is probably closest to our approach. It focuses on the schedulability analysis of real-time systems but supports static architectures, only. SOFA HI is an extension of the SOFA component model for real-time systems. It provides extensive reuse capabilities and development support as well as the possibility to change the inner structure of a component at runtime. Additionally, SOFA HI supports checking compliance of behavior protocols and the component implementation. However, that does not include 1:n communications with dependencies or relaxation of time intervals. The Progress component model targets automotive applications and provides passive, signal based components which can be structured hierarchically. Thus, the concepts cannot be applied directly to our active, message based components.

In the area of graph transformations, [7] defines a refinement for hybrid graph transformation systems which preserves verification results of the abstract behavior. The focus is not, as in our case, to define a more relaxed refinement which enables a more flexible integration of possible refined behavior and it is not required that the external visible real-time behavior is still preserved by the refined behavior. [12] considers graph transformation systems for the specification of service oriented architectures. The presented refinement should preserve the external visible services. The approach did not take into account time and the ability to preserve verification results. [11] examine refinement for graph transformation systems based on an algebra but they did not take into account time.

For (timed) automata, there exist several refinement definitions. In [4], simulation and bisimulation are defined. Simulation only preserves properties containing only \( \forall \) quantifications and does not require that the refined system has the same externally visible behavior. Both properties are fulfilled for bisimulation, but it is more restrictive than our refinement definition as defined in Section 3. Both kinds of refinement relation have been extended to real-time systems in [23]. In [18], a refinement check for the model checker UPPAAL supporting urgent synchronizations is presented. The algorithm is based on so-called test automata and requires one reachability analysis on a product automaton of system-to-test and test automaton, only, to show a correct refinement. The presented algorithm, however, is not applicable to reconfigurable systems in its current form.

6. CONCLUSION AND FUTURE WORK

In this paper, we introduced a refinement definition and a refinement check. The refinement preserves safety and bounded liveness properties as well as the externally visible real-time behavior of a self-adaptive system. Our refinement check is based on a reachability analysis on Timed Graph Transformation Systems. Our presented refinement approach also supports a time relaxation. Hereby, our approach supports the refinement of the behavior in a more flexible way than standard refinement definitions like bisimulation. Our evaluation results emphasize that checking for correct refinements is more efficient than re-verifying the refined behavior specification.
Since different application domains, e.g., automotive systems, may require other time relaxations, we intend to extend our refinement definition. The extension is a parameterization of our refinement definition such that different refinement definition can be obtained by assigning the parameters accordingly. That allows to support a family of similar refinement definition in one definition. Consequently, we plan to adapt our refinement check to this parameterized definition.

Acknowledgments

This work was developed in the course of the Collaborative Research Center 614 – Self-optimizing Concepts and Structures in Mechanical Engineering – University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft.

Christian Heinzemann is supported by the International Graduate School Dynamic Intelligent Systems.

We thank Steffen Becker for his valuable comments on draft versions of the paper.

7. REFERENCES