Abstract: The trend towards more complex software within today’s technical systems results in an increasing demand for dependable high quality software for real-time systems. In this report the foundations for the compositional pattern-based design of correct high level designs and architectures for real-time systems are presented. A formal calculus including a notion of discrete-time automata, a deadlock preserving refinement notion, and a class of supported compositional constraints is developed. It permits to design the required complex cooperation between the system components using verified patterns and includes support to derived the related correct component behavior in a systematic manner, such that the components itself do not invalidate the verified cooperation patterns.

keywords: Real-Time Systems, Compositional Verification, Model Checking, Behavior Synthesis
## Contents

1 Introduction 3  
2 Related Work 4  
3 A Compositional Discrete-Time Calculus 5  
   3.1 Model of Behavior 5  
   3.1.1 Formal Model 6  
   3.1.2 Characteristics 10  
   3.2 Refinement 11  
   3.2.1 Definitions 11  
   3.2.2 Characteristics 14  
   3.2.3 Ensuring Refinement 16  
   3.3 Compositional Reasoning 18  
   3.3.1 Definitions 18  
   3.3.2 Characteristics 19  
4 Pattern-Based Real-Time System Modeling 20  
   4.1 Patterns, Components and System 21  
   4.1.1 Pattern Definition 21  
   4.1.2 Component Definition 22  
   4.1.3 System Definition 23  
   4.2 Compositional Reasoning 25  
   4.2.1 Local Properties 25  
   4.2.2 Non Local Properties 27  
   4.2.3 Incremental Checking 29  
   4.2.4 Checking and Changes 30  
   4.3 Design and Verification Steps 30  
   4.3.1 Pattern Design 30  
   4.3.2 Component Design 30  
   4.4 Sequence of Design and Verification Steps 32  
5 Conclusion and Future Work 33
1 Introduction

The trend towards more complex software within today’s technical systems results in an increasing demand for high quality software for real-time systems. In many of these systems such as mechatronic systems their safety-critical nature requires that the embedded software does not result in an overall system dependability which is below the level which has been reached by former systems. However, as the current trend results in more complex embedded software and beforehand isolated operating software is often interconnected via networks, a rapid growth of the system complexity can be observed. Therefore, serious software bugs in such systems are actually more likely than in more traditionally developed systems that contain less software and are less complex.

A number of techniques for dependable system design exist to recover from hardware or software failures. However, these approaches are mainly limited to address local design or coding errors which can be detected and compensated by appropriate local recovery mechanisms. Design errors at the architectural level in contrast only show up when multiple system components interact, the options for detection and recovery of these errors are rather limited.

The verification of the correctness of the overall design is besides other important activities such as careful analysis and testing only one crucial prerequisite for dependable software. The complexity of software and the rather arbitrary nature of its faults renders attempts to dependable systems without a correct high level design a hopeless undertaking. The correctness of the higher level software design and architecture, which includes appropriate means to compensate likely hardware or component failures that have not been already addressed locally, is therefore usually one cornerstone of any good engineered complex dependable real-time system.

In highly safety-critical real-time systems, today most often an architecture is chosen that by itself reduces the resulting system complexity as much as possible by decoupling the different system components in such a manner, that safety or other required dependability attributes can be realized by the local subsystems. However, this usually results in quite high system costs and limits the introduction of more sophisticated system functionality or cooperation via the network.

The presented concepts for the compositional pattern-based design of correct high level designs and architectures for real-time systems propose a formal framework for compositional design and reasoning to overcome the outlined limitations.

A formal calculus with a discrete notion of timed automata, a refinement notion that preserves deadlock freedom, and a classification of properties that can be verified in a compositional manner build the underlying formal framework. This formal framework further permits to design the required complex cooperation between the system components using verified patterns and therefore helps to exclude that the cooperation itself can be erroneous. The component behavior required by the system context can be further derived in a systematic manner, which ensures that the components itself do not invalidate the verified cooperation patterns. The combination of both steps and the syntactical correct composition of the overall system does then ensure that the verified properties for the cooperations as well as components also hold for the overall system.

Moreover, for the local verification steps of the proposed approach, which are required to either proof the correctness of a pattern or a component, the integration of model checking to automate the tedious verification tasks is presented. Therefore, the cost for correctness can be
reduced to a great extend.

If reasonable component or network failures are also reflected within the model, the compositional verification ensures that the specific high level design and architecture operates correct even for those failures. Therefore, when additionally exploiting the potential for reuse related to the notion of pattern and components for domain and technology specific coordination protocols, the presented work lays the foundation for the systematic correct high level design of complex dependable systems.

The rest of the report is organized as follows: We first review some related work about compositional verification and especially compositional model checking in Section 2. In Section 3 the underlying formal model and main results for compositional reasoning are presented. Their application for the correct compositional construction and verification of real-time system by means of patterns and components is then presented in Section 4. The report is closed in Section 5 with some concluding remarks, a review of the remaining limitations and an outlook on future work.

2 Related Work

To address the compositional verification of the high level design of real-time systems, we have to look for model checking techniques that support an explicit notion of time. In this context, two different approaches in the literature can be distinguished.

A first thread of work extends the standard model checking framework developed for the future-oriented branching-time Computation Tree Logic (CTL, see [CGP00]). The transitions of the untimed models are simply interpreted as time steps w.r.t. a discrete and global notion of time. For the specification of timed properties a number of extensions of CTL have been developed, e.g., RTCTL (real time CTL [EMSS92]) or CCTL (clocked CTL [Ruf01]). The tools which follow this direction are VERUS [CCM97] and RAVEN [Ruf01].

Another approach that considers a continuous notion of time are timed automata by Alur et al. [ACD90]. In timed automata, an arbitrary finite number of real-value clocks can be employed to measure time differences between clock resets and measurements w.r.t. a global continuous notion of time. Timed CTL (TCTL), an extension of CTL with dense-time semantics, can be used to specify required properties. Tools that support timed automata are KRONOS [Yov97] and UPPAAL [BBD02].

In general, model checking of higher level software models is limited due to the state explosion problem, which leads to scalability problems for larger systems even when no time is considered (cf. [CAB98]). A number of modular and compositional verification approaches have therefore been proposed.

In [LKF02], similar to the presented approach the decomposition of a system is exploited to permit modular model checking of the system. The notion of decomposition into features in [LKF02] is limited to the sequential case and thus does not address the classical state explosion problem. It cannot be employed for complex real-time systems, because besides sequential also the parallel composition of components is required.

One particular general approach for the compositional verification is the assume/guarantee
paradigm [MC81]. Model checking techniques that permit compositional verification following
the assume/guarantee paradigm have been developed [CGP00, p. 185ff].

The presented approach also follows the assume/guarantee paradigm, but in contrast to cur-
current proposals exploits information available during the design process in form of overlapping
protocols to derive the required additional assumed and guaranteed properties automatically
rather than manually as in [CGP00]. Moreover, we employ a more restricted notion of refinement
which also excludes deadlocks, whereas in [CGP00] only a subset of CTL restricted to only \( A \)
path quantifier in the positive normal form (called ACTL) can be applied. Note that ACTL cannot
be used to compositionally exclude deadlocks, because deadlock freedom is \( (AG(\exists X true)) \)
in CTL and thus not an ACTL formula.

3 A Compositional Discrete-Time Calculus

In this section the formal foundations for the compositional reasoning for real-time systems is
presented. This includes at first the description of the employed behavior model in Section 3.1.
Then, in Section 3.2 the behavior refinement notion developed for the compositional reasoning
is defined. The additionally developed techniques for compositional reasoning follow in Section
3.3.

3.1 Model of Behavior

As a correct high level design and architecture are one goal of our approach, it is reasonable
to focus on the dynamic model and the synchronization skeleton (cf. [EC82]) of the system
components to exclude design errors at this level of abstraction. Therefore, we can assume
that the component behavior can (at least in an abstract fashion) be described by an real-time
extension of a finite automata which can in principle be subject to verification by model checking.

Both general approaches for model checking real-time systems [EMSS92, Ruf01, CCM97,
ACD90, Yov97, BBD+02] to some extent assume that all employed clocks run with the same
speed. Therefore, both assume a global clock. We therefore cannot avoid this assumption and
the related problems when we want to employ any of them.

In general, if multiple automata operate with the same period their synchronization can ex-
clude or ensure certain properties which do not hold for the real system due to clock skews. To
limit this problem, we thus restrict the employed models to such ones which do not directly syn-
chronize automata which in the real model may later operate on different hosts. Instead, special
automata describing the possible delay of the connection media (network) are assumed. As these
glue behavior explicitly includes possible delays the assumption of a global clock is reduced to
the assumption that the clock screw between different hosts has a relative small upper bound and
any resulting effect is fully covered by the delay effects of the communication media.

When these model restriction has been made, both the discrete-time and continuous time
approaches would in general permit to model the real-time behavior of real systems. As the
discrete-time model is considerable simpler, better permits to describe the required notions of
refinement, and thus ease to prove the required compositionallity result, we employ in this report
the discrete-time model. However, we expect that the presented results can be transferred at least to such continuous-time models where a discrete abstraction exists.

In this section we first describe the formal model used for the description of the operation real-time behavior in form of automata as well as real-time properties in an appropriate real-time temporal logic in Section 3.1.1. Then, some rather fundamental characteristics of such models are summarized and proven in Section 3.1.2.

3.1.1 Formal Model

The following notion of a discrete real-time automaton is used to describe the required real-time behavioral of a systems and its elements.

**Definition 1.** A simple real-time automaton is a 5-tuple \( M = (S, I, O, T, Q) \) with a finite set \( S \) of states, a finite set \( I \) of input signals, a finite set \( O \) of output signals, a finite set of transitions \( T \subseteq S \times \wp(I) \times \wp(O) \times S \), and the initial state set \( Q \).

The time semantics of an automaton is simply that the execution of each transition takes exactly one time unit. The notion of a run further denotes a possible sequence of states which may be observed during the execution of the automata.

**Definition 2.** A run of a simple real-time automaton \( M = (S, I, O, T, Q) \) is a sequence of states \( \langle s_1, s_2, \ldots \rangle \), where for each \( i \geq 1 \) exists \( (s_i, A, B, s_{i+1}) \in T \).

We further require that a real-time automaton only contains reachable states within its state set.

**Definition 3.** A simple real-time automaton \( M = (S, I, O, T, Q) \) is well-formed iff for each \( s \in S \) there is at least one finite run \( \langle s_1, s_2, \ldots, s_n \rangle \) with \( s_1 \in Q \) and \( s = s_n \).

All further discussions are restricted to well-formed real-time automata. This does not exclude any possible behavior as non reachable states do obviously not contribute to the resulting behavior.

For convenience we introduce the following notation and definitions: The symbols \( S, I, O, T, Q \) are used to denote the corresponding elements of \( M_i \). Two automata \( M \) and \( M' \) with distinct input and output sets \((I \cap I' = \emptyset \text{ and } O \cap O' = \emptyset)\) are further called **composable**. If also \( I \cap O' = \emptyset \text{ and } O \cap I' = \emptyset \) holds, they are even **orthogonal** to each other. We call the signals contained in \( I \cap O \) the **internal signals** and further classify an automaton \( M \) with \( I = O \) as **closed**.

**Property Specification**

To reflect the dependencies which result from local or shared name spaces within property specifications, a shared set of atomic propositions \( P \) is used to describe basic properties. An automaton \( M_i \) is equipped with its property set \( P_i \subseteq P \) and any of its states \( s \in S_i \) is annotated with all that propositions in \( P_i \) which \( s \) fulfills using a labelling function \( L_i : S \to \wp(P_i) \). Thus an automaton \( M_i = (S_i, I_i, O_i, T_i, Q_i) \) is accordingly extended to a 7-tuple \( M_i = (S_i, I_i, O_i, T_i, P_i, L_i, Q_i) \).
Definition 4. A real-time automaton is a 7-tuple \( M = (S, I, O, T, P, L, Q) \) for which holds that \( M' = (S, I, O, T, Q) \) is a well-formed simple real-time automaton, \( P \subseteq P \) is the label set, and \( L : S \rightarrow \wp(P) \) is the related labelling function.

The label set \( P \) is further denoted by the labelling of the automata \( M_i \) itself using \( L(M_i) \).

Note that disjoint labelling sets of multiple automata can be used to emulate the effects of local name spaces. Shared naming can be employed when consistent labelling between different automata is required.

Using the basic set \( P \) of atomic propositions we are able to specify which properties should hold for a specific model using a state-oriented temporal logic called *Clocked CTL (CCTL)* [RK99]. Therefore, in CCTL we also have the \( A \) all and \( E \) exists path quantor of CTL. In contrast to classical CTL [CGP00], the temporal operators \( F \) (eventually), \( G \) (globally), \( U \) (until), and \( R \) (release) are defined for intervals \( [a, b] \), \( a \in \mathbb{N}_0, b \in \mathbb{N}_0 \cup \{\infty\} \). The \( X\)-operator (next) has a single time-bound \( [a] \) with \( a \in \mathbb{N} \).

Definition 5. For a given basic set of atomic propositions \( P \subseteq P \) all valid CCTL formulas are defined as follows:

- all \( p \in P \) and the constants \( \text{true} \) and \( \text{false} \) are valid CCTL formulas,
- for two valid CCTL formulas \( \phi \) and \( \phi' \) also \( \phi \land \phi' \) and \( \neg \phi \) are valid CCTL formulas,
- for a valid CCTL formula \( \phi \) also \( \square[a] \phi \) with \( \square \in \{EX, AX\} \) and \( a \in \mathbb{N} \) are valid CCTL formulas,
- for a valid CCTL formula \( \phi \) also \( \square[a,b] \phi \) with \( \square \in \{EF, AF, EG, AG\} \) and \( a \in \mathbb{N}_0, b \in \mathbb{N}_0 \cup \{\infty\} \) are valid CCTL formulas, and
- for two valid CCTL formulas \( \phi \) and \( \phi' \) also \( \square(\phi \ U_{[a,b]} \phi') \) and \( \square(\phi \ R_{[a,b]} \phi') \) with \( \square \in \{A, E\} \) and \( a \in \mathbb{N}_0, b \in \mathbb{N}_0 \cup \{\infty\} \) are valid CCTL formulas.

Other standard operators of propositional logic such as \( \lor \), or \( \Rightarrow \) are mapped to combinations of negation and \( \land \) like in the propositional case.

The semantics of CCTL is defined by the ”\( \models \)” operator which refers to an initial state and related runs of a real-time automaton \( M \).

Definition 6. For valid CCTL formulas \( \phi \) and \( \psi \), an atomic proposition \( p \), a model \( M = (S, I, O, T, P, L, Q) \) in form of a real-time automaton, and \( s_0 \) a state of \( M \) the following rules defines when the \( \models \) operator hold:

- \( M, s_0 \models p \) with \( (p \in P) \) iff \( p \in L(s_0) \).
- \( M, s_0 \models \neg \phi \) iff \( M, s_0 \models \phi \) is false.
- \( M, s_0 \models (\phi \land \psi) \) iff \( M, s_0 \models \phi \) and \( M, s_0 \models \psi \).
- \( M, s_0 \models EX_{[a]} \phi \) iff there exists a run \( r = (s_0, \ldots, s_a, \ldots) \) such that \( M, s_a \models \phi \) (exists next).
• $M, s_0 \models \text{EF}_{[a,b]} \phi$ iff there exists a run $r = \langle s_0, \ldots, s_a, \ldots, s_b, \ldots \rangle$ and some $i$ with $a \leq i \leq b$ such that $M, s_i \models \phi$ (exists eventually).

• $M, s_0 \models \text{EG}_{[a,b]} \phi$ iff there exists a run $r = \langle s_0, \ldots, s_a, \ldots, s_b, \ldots \rangle$ such that for all $i$ with $a \leq i \leq b$ holds $M, s_i \models \phi$ (exists globally).

• $M, s_0 \models \text{E}(\phi \cup_{[a,b]} \psi)$ iff there exists a run $r = \langle s_0, \ldots, s_a, \ldots, s_b, \ldots \rangle$ and some $i$ with $a \leq i \leq b$ such that $M, s_i \models \psi$ and for all $j < i$ holds $M, s_j \models \phi$ (exists until).

• $M, s_0 \models \text{E}(\phi \mathcal{R}_{[a,b]} \psi)$ iff there exists a run $r = \langle s_0, \ldots, s_a, \ldots, s_b, \ldots \rangle$ with for all $a \leq j \leq b$ holds that if for every $a \leq i \leq j$ holds $M, s_i \models \phi$ then $M, s_j \models \psi$ (exists release).

• $M, s_0 \models \text{AX}_{[a]} \phi$ iff for all runs $r = \langle s_0, \ldots, s_a, \ldots, s_b, \ldots \rangle$ hold that $M, s_a \models \phi$ (all next).

• $M, s_0 \models \text{AF}_{[a,b]} \phi$ iff for all runs $r = \langle s_0, \ldots, s_a, \ldots, s_b, \ldots \rangle$ there exists some $i$ with $a \leq i \leq b$ such that $M, s_i \models \phi$ (all eventually).

• $M, s_0 \models \text{AG}_{[a,b]} \phi$ iff for all runs $r = \langle s_0, \ldots, s_a, \ldots, s_b, \ldots \rangle$ and all $i$ with $a \leq i \leq b$ holds $M, s_i \models \phi$ (all globally).

• $M, s_0 \models \text{A}(\phi \cup_{[a,b]} \psi)$ iff for all runs $r = \langle s_0, \ldots, s_a, \ldots, s_b, \ldots \rangle$ there exists some $i$ with $a \leq i \leq b$ such that $M, s_i \models \psi$ and for all $j < i$ holds $M, s_j \models \phi$ (all until).

• $M, s_0 \models \text{A}(\phi \mathcal{R}_{[a,b]} \psi)$ iff for all runs $r = \langle s_0, \ldots, s_a, \ldots, s_b, \ldots \rangle$ with for all $a \leq j \leq b$ holds that if for every $a \leq i \leq j$ holds $M, s_i \not\models \phi$ then $M, s_j \models \psi$ (all release).

The classical non real-time temporal operators can simply be defined by using $[1]$ and $[0, \infty]$ as default for the time intervals. $\mathcal{L}(\phi)$ further denotes the subset of the basic proposition set $\mathcal{P}$ that is employed in the formula $\phi$.

The subset of CCTL which only uses the all path quantifier and has negation only in the propositions (positive normal form) is further called ACCTL.

**Definition 7.** For a given basic set of atomic propositions $P$ all valid ACCTL formulas are defined as follows:

• the constants true and false and for $p \in P$ $p$ as well as $\lnot p$ are valid ACCTL formulas,

• for two valid ACCTL formulas $\phi$ and $\phi'$ also $\phi \land \phi'$ and $\phi \lor \phi'$ are a valid ACCTL formulas,

• for a valid ACCTL formula $\phi$ also $\text{AX}_{[a]} \phi$ with $a \in \mathbb{N}$ is a valid ACCTL formula,

• for a valid ACCTL formula $\phi$ also $\Box_{[a]} \phi$ with $\Box \in \{\text{AF}, \text{AG}\}$ and $a \in \mathbb{N}_0, b \in \mathbb{N}_0 \cup \{\infty\}$ are valid ACCTL formulas, and

• for two valid ACCTL formulas $\phi$ and $\phi'$ also $A(\phi \cup_{[a,b]} \phi')$ and $A(\phi \mathcal{R}_{[a,b]} \phi')$ with $a \in \mathbb{N}_0, b \in \mathbb{N}_0 \cup \{\infty\}$ are valid ACCTL formulas.
Finally, for sake of simplification of the following formal definitions, we will often omit the syntactical details of CCTL and write $M \models \phi$ when an automaton $M$ fulfills a constraint $\phi$ for all its initial states (for all $s \in Q$ hold $M, s \models \phi$). For invariants we have $M \models \psi$ simply when for all $s \in S$ holds $M, s \models \psi$.

A property of special interest is the deadlock. A state is a deadlock iff the state has no outgoing transition. This can be characterized in CCTL as $\neg(\text{EX true})$. The special symbol $\delta$ is used to denote that such a deadlock state can be reached, which therefore is in CCTL $\neg(\text{AG (EX true)})$. $M \models \neg\delta$ thus denotes that $M$ is free from deadlocks, can be expressed in CCTL normal form by $\text{AG (EX true)}$. Thus, deadlock freedom is not an ACCTL formula. Note that we further define that any automaton $M_i$ with empty state set ($S_i = \emptyset$) contains a deadlock.

**Restriction, Hiding and Relabelling**

Within the presented formal model signals are used to describe the synchronization of a real-time automaton with its environment. A restriction operator can be further used to abstract from signals where required.

**Definition 8.** For an automaton $M = (S, I, O, T, P, L, Q)$ we define its I/O restriction for $I''/O''/P''$ denoted by $M|_{I''/O''/P''}$ as the automaton $(S', I', O', T', P', L', Q')$ with $S' = S$, $I' = I \cap I''$, $O' = O \cap O''$, $P' = P \cap P''$, $L'(x) = L(x) \cap P''$, $Q' = Q$, and $(s_1, A', B', s_2) \in T'$ iff $(s_1, A, B, s_2) \in T$ exists with $A' = A - I''$ and $B' = B - O''$.

Using the restriction operator, we further define the *hiding* $M_i\setminus_{I/O/P} M_i|_{I_i-I/O_i-P}$.

A relabelling operator $[]$ together with a relabelling $\alpha$ can be further defined to systematically exchange the defined I/O signals and proposition labels such that multiple instances of a real-time automata can be derived from a single definition.

**Definition 9.** For an automaton $M = (S, I, O, T, P, L, Q)$ we define its relabelling for $\alpha = (\alpha_I, \alpha_O, \alpha_L)$ denoted by $M[\alpha]$ as the automaton $(S', I', O', T', P', L', Q')$ with $S' = S$, $I' = \{\alpha_I(x) | x \in I\}$, $O' = \{\alpha_O(y) | y \in O\}$, $P' = \{\alpha_L(p) | p \in P\}$, $L' = \alpha_L \circ L$, $Q' = Q$, and $(s_1, A', B', s_2) \in T'$ iff $(s_1, A, B, s_2) \in T$ exists with $A' = \{\alpha_I(x) | x \in A\}$ and $B' = \{\alpha_O(y) | y \in B\}$.

**Parallel Composition**

As we model time explicitly and in a discrete manner, the required notion of parallel composition must result in the *synchronous execution* [CGP00] of all systems running in parallel.

The communication via signals is formalized by *synchronous communication* such that sending and receiving happens within the same time step. Consequently, when asynchronous event semantics such as for real-time statecharts [GB03] has to be modelled event queues (channels) have to be explicitly modelled in form of additional automata. It is to be noted, that such explicit models of the event queues are required anyway when quality of service (QoS) characteristics of each connection have to be taken into account.
To combine two composable automata we simply connect their input and output signals and consider their synchronous parallel execution.

**Definition 10.** For two automata \( M = (S, I, O, T, P, L, Q) \) and \( M' = (S', I', O', T', P', L', Q') \) which are composable to each other, we define their parallel composition denoted by \( M \parallel M' \) as the automaton \( (S'', I'', O'', T'', P'', L'', Q'') \) with \( S'' = S \times S' \), \( I'' = I \cup I' \), \( O'' = O \cup O' \), \( Q'' = Q \times Q' \), and \( (s_1, s_1'), A'', B'', (s_2, s_2') \in T'' \) iff \((s_1, A, B, s_2) \in T \) and \((s_1', A', B', s_2') \in T' \) exist with \( A'' = A \cup A' \), \( B'' = B \cup B' \) and \( \mathcal{L}(M) \cap L''(s_2') = \mathcal{L}(M') \cap L(s_2) \). Additionally, \((A \cap O') = B' \) and \((A' \cap O) = B \) must hold. \( S'' \) and \( T'' \) are further adjusted to exclude all non reachable state combinations and transitions to ensure that the resulting automaton is well-formed. \( P'' = P \cup P' \) and the labelling \( L'' \) for \((s, s') \in S'' \) is derived as \( L''((s, s')) = L(s) \cup L'(s') \).

Informally, a transition in \( T'' \) is a combination of two transitions in each automaton iff all required local inputs by the other side are matching ((\( A \cap O' \) = \( B' \) and \( A' \cap O \) = \( B \)) and the shared labelling of the target states is identical. It is important to note, that for shared labels, the combined resulting states have to be labelled identically (\( \mathcal{L}(M) \cap L'(s_2') = \mathcal{L}(M') \cap L(s_2) \)) or will be discarded otherwise.\(^1\) The external input and output signals are simply the union of both automata.

If we want to abstract from the internal I/O we simply use the restriction for \( I_1/O_1 \) with \( I_1 = I'' - (O \cup O') \) and \( O_1 = O'' - (I \cup I') \).

Note that the employed discrete modeling of time would also imply that a deadlock describes that time cannot further elapse (time stopping deadlock) and thus is a failure in the model itself w.r.t. time. For the parallel composition, for example, a time deadlock reflects that the time constraints of the two composed automata are not compatible. Consequently, an empty automata which might result from a parallel composition contains by definition a deadlock.

### 3.1.2 Characteristics

For the introduced formal model of discrete-time automata, the following rather trivial characteristics that the local constraints of the automata are preserved by parallel composition when the labelling is disjoint, can be proven as follows:

**Lemma 1.** For automata \( M_1 \) and \( M_2 \) and constraints \( \phi_1 \) and \( \phi_2 \) with \( I_1 \cap O_2 = \emptyset \), \( O_1 \cap I_2 = \emptyset \), \( \mathcal{L}(M_1) \cap \mathcal{L}(M_2) = \emptyset \) holds

\[
((M_1 \models \phi_1) \land (M_2 \models \phi_2)) \Rightarrow (M_1 \parallel M_2 \models \phi_1 \land \phi_2)
\]

**Proof.** (sketch) As \( \mathcal{L}(M_1) \cap \mathcal{L}(M_2) = \emptyset \), we can conclude that whether \( \phi_2 \) holds is not influenced by \( M_1 \). Due to the assumptions \( I_1 \cap O_2 = \emptyset \), \( O_1 \cap I_2 = \emptyset \), and \( \mathcal{L}(M_1) \cap \mathcal{L}(M_2) = \emptyset \) both automata are executed independently in parallel and thus \( M_2 \models \phi_2 \) will imply \( M_1 \parallel M_2 \models \phi_2 \). We can argue analogously for \( \phi_1 \) and thus have proven the above result. \( \square \)

\(^1\)It is to be noted, that this condition will erase certain state combinations by referring to the labelling of each of the local state spaces. Therefore, the combination of real-time automata with shared labels does to some extent not respect their local behavior when they are seen as systems interacting only via communication. Therefore, later in Section 4 we will only combine real-time automata with overlapping constraints within a single component where such sharing of knowledge is reasonable.
3.2 Refinement

The central notion of a property preserving modification is refinement. In the following we will define a set of different notions of refinement in Section 3.2.1. Then, in Section 3.2.2 some later required characteristics of the introduced refinement notions are presented. Transformation rules for refinement, an algorithm to check the presented refinement notions and a sketch for a synthesis algorithm for the maximal refining behavior w.r.t. a given invariant are presented in Section 3.2.3.

3.2.1 Definitions

For the defined notion of real-time automata an appropriate notion for refinement has to ensure two fundamental properties. (1) We require that each behavior of the refining behavior can be observed in the original one and (2) if the original behavior offers a transition with specific input and output signals the refining behavior must also offer the same. Otherwise a specific testing environment exists where the original behavior will succeed while the refinement will not. For more details on the underlying phenomena see the CSP failure divergence refinement [Hoa85] or testing theory in general [vG90, vG93].

In the following we review some fundamental notions of behavioral relations and discuss what is missing to fulfill the above requirements.

Simulation

A structural notion of simulation [Mil71] is defined as follows:

**Definition 11.** For automata \( M = (S, I, O, T, P, L, Q) \) and \( M' = (S', I', O', T', P', L', Q') \) we call \( M \) a simulation of \( M' \) denoted by \( M \preceq M' \) iff a relation \( \Omega \subseteq S \times S' \) exists with \( \forall q \in Q \exists q' \in Q' : (q, q') \in \Omega \) and for all \( (s_1, s'_1) \in \Omega \) holds:

\[
\forall (s_1, A, s_2) \in T \quad \exists (s'_1, A, s'_2) \in T' : (s_2, s'_2) \in \Omega.
\]

For the given labellings we require \( P = P' \) and that \( L \) and \( L' \) are preserved by \( \Omega \): \( (s, s') \in \Omega \Rightarrow L(s) = L'(s') \).

The relation \( \Omega \) initially ensures that for each initial state of the simulating automaton an appropriate interpretation in terms of the initial state of the simulated automaton exists. While simulation result in a partial ordering of different automata, it is too weak to ensure that deadlock freedom is preserved under substitution.

Bisimulation

If we adjust the structural notion of simulation such that they hold in both directions at once, we have bisimulation [Mil89], which can be defined as follows:
Definition 12. For automata \( M = (S, I, O, T, P, L, Q) \) and \( M' = (S', I', O', T', P', L', Q') \) we call \( M \) bisimilar to \( M' \) denoted by \( M \cong M' \) iff a relation \( \Omega \subseteq S \times S' \) exists with \( \forall q \in Q \exists q' \in Q' : (q, q') \in \Omega \) and for all \( (s_1, s'_1) \in \Omega \) holds:

\[
\forall (s_1, A, B, s_2) \in T \exists (s'_1, A, B, s'_2) \in T' : (s_2, s'_2) \in \Omega,
\]

(2) \[
\forall (s'_1, A, B, s'_2) \in T' \exists (s_1, A, B, s_2) \in T : (s_2, s'_2) \in \Omega,
\]

We require that \( L \) and \( L' \) are preserved by \( \Omega \): \( (s, s') \in \Omega \Rightarrow L(s) = L'(s') \) and that \( P = P' \).

The relation \( \Omega \) initially ensures that for each state pair a state pair and possible transitions with equal I/O signals again a state pair can be reached which is identically labelled and structurally equivalent. Therefore, bisimulation is an equivalence relation which is too strong as the required degrees of freedom to further "refine" the behavior are not given.

Refinement

As argued before, for the defined notion of real-time automata an appropriate notion for refinement has to ensure two fundamental properties. (1) We require that each behavior of the refining behavior can be observed in the original one and (2) if the original behavior offers a transition with specific input and output signals the refining behavior must also offer a transition with the same I/O signal sets to ensure that deadlock freedom is preserved. When both requirements are fulfilled, a notion of refinement which is strong enough to preserve deadlock freedom and weak enough to permit further refinements results.

The employed basic structural notion of refinement that fulfill the above described requirements is defined as follows:  

Definition 13. For automata \( M = (S, I, O, T, P, L, Q) \) and \( M' = (S', I', O', T', P', L', Q') \) we call \( M \) a refinement of \( M' \) denoted by \( M \sqsubseteq M' \) iff a relation \( \Omega \subseteq S \times S' \) exists with \( \forall q \in Q \exists q' \in Q' : (q, q') \in \Omega \) and for all \( (s_1, s'_1) \in \Omega \) holds:

\[
\forall (s_1, A, B, s_2) \in T \exists (s'_1, A, B, s'_2) \in T' : (s_2, s'_2) \in \Omega,
\]

(4) \[
\forall (s'_1, A, B, s'_2) \in T' \exists (s_1, A, B, s_2) \in T : (s_2, s'_2) \in \Omega,
\]

For a given labelling we require \( P = P' \) and that \( L \) and \( L' \) are preserved by \( \Omega \): \( (s, s') \in \Omega \Rightarrow L(s) = L'(s') \).

The relation \( \Omega \) initially ensures that for each initial state of the refinement an appropriate interpretation in terms of the initial state of the refined automaton exists. For each transition in the refinement \( M \) equation 4 further ensures that a related transition in \( M' \) exists that again

\footnote{As an overwhelming number of proposals for refinement/abstraction in the area of process algebras have been proposed (cf. [vG90, vG93]), we do not claim to invent this specific one even though we are not aware of any identical proposal. As the timed interpretation of transition system is not standard for process algebras, at least our interpretation will differ substantially from any refinement notion for standard process algebras.}
leads to an appropriate state pair in $\Omega$ like condition 1 for simulation. Therefore, refinement ($\sqsubseteq$) implies simulation ($\preceq$).

Equation 5 further ensures that for each pair of I/O signal sets offered by a state in $M'$ a corresponding transition offering the same pair of I/O signal sets is provided in its refinement $M$. However, the condition does not itself require that $s_3$ and $s'_3$ build a pair contained in $\Omega$.

It is to be noted, that the conditions for a refinement assume not only a behavioral refinement w.r.t. the signals and labelling but also a structural similarity. A structure independent notion of refinement has to employ the more abstract concepts of traces and failures (cf. CSP failure divergence refinement [Hoa85]) instead of comparing the transition structures itself. This restricts the general application for two arbitrary automata. In our settings, however, this reduced notion will be sufficient, because in our approach we will only require that a refinement relation has to be established where also structural similarity naturally follows from the design process.

3.2 Refinement

I/O Refinement

It is to be noted that we expect from a refinement notion a certain degree of freedom to also extend the refined behavior. To have a refinement notion that permits the refined behavior to extend the original one, we have to combine refinement with abstraction using the restriction operator $\mid$. If for two basic automata $M$ and $M'$ hold that $M$ restricted to the input, output, and labelling of $M'$ is a refinement of $M'$, we further call $M$ an I/O refinement of $M'$.

**Definition 14.** For automata $M = (S, I, O, T, P, L, Q)$ and $M' = (S', I', O', T', P', L', Q')$ we name $M$ an I/O refinement of automaton $M'$ denoted by $M \sqsubseteq_{I/O} M'$ iff $M |_{P' \cap O'/P'} \sqsubseteq M'$.

The I/O refinement adjusts the considered signals and can be further used to characterize if an automaton is a correct concretization of another one.

Restricted I/O Refinement

A more restricted version of refinement can be derived which additionally requires that only purely internal states and transitions are extended w.r.t. the signal sets. A state $s \in S$ of an automaton $M$ is therefore classified as *internal* denoted by $s \in \overline{S}$ iff all its transitions are only internal ones ($\forall (s, A, B, s') \in T$ holds $A = B = \emptyset$). Using this notion we have:

**Definition 15.** For automata $M = (S, I, O, T, P, L, Q)$ and $M' = (S', I', O', T', P', L', Q')$ we call $M$ a restricted I/O refinement of $M'$ denoted by $M \sqsubseteq^*_{I/O} M'$ iff a relation $\Omega \subseteq S \times S'$ exists with $\forall q \in Q \exists q' \in Q': (q, q') \in \Omega$ and for all $(s_1, s'_1) \in \Omega$ holds:

$$s'_1 \in \overline{S} \Rightarrow \forall (s_1, A, B, s_2) \in T \exists (s'_1, \emptyset, \emptyset, s'_2) \in T': (s_2, s'_2) \in \Omega \wedge A \subseteq I \wedge B \subseteq O - O', \quad (6)$$

$$s'_1 \notin \overline{S} \Rightarrow \forall (s_1, A, B, s_2) \in T \exists (s'_1, A, B, s'_2) \in T': (s_2, s'_2) \in \Omega, \quad (7)$$

$$\forall (s'_1, A', B', s'_3) \in T' \exists (s_1, A', B', s_3) \in T. \quad (8)$$

$P = P'$ must hold and for the given labelling functions $L$ and $L'$ we require again that they are preserved by $\Omega$: $(s, s') \in \Omega \Rightarrow L(s) = L'(s')$. 

13
For restricted I/O refinement holds that the equations 6 and 7 are somehow strengthening equation 4 from the definition of refinement while equation 8 is identical to equation 5. Therefore, it holds that restricted I/O refinement implies I/O refinement.

3.2.2 Characteristics

For the presented refinement notions we are able to prove the following fundamental results which will be later employed to enable compositional reasoning.

Refinement Preserves Deadlock Freedom

CTL formulas are preserved by the bisimulation equivalence relation, while ACTL formulas are preserved by the simulation preorder ($\preceq$) [CGP00]. The presented refinement implies simulation and thus preserves ACTL formulas resp. its clocked version ACCTL also, but in contrast it additionally preserves deadlock freedom:

**Lemma 2.** For automata $M$ and $M'$ with $M \sqsubseteq M'$ holds $M' \models \neg \delta \Rightarrow M \models \neg \delta$.

**Proof.** If not $M' \models \neg \delta \Rightarrow M \models \neg \delta$, we can conclude that a refinement $M$ which contains a deadlock state $s \in S$ and an original behavior $M'$ which does not contain any deadlock state exists. Due to condition 4 we can identify any reachable state $s$ and thus also at least one state $s' \in S'$ with $(s, s') \in \Omega$ (the relation $\Omega$ implied by the assumed refinement). As $M'$ does not have a deadlock state, $s'$ has at least one outgoing transition and thus due to condition 5 also $s$. This contradicts our initial assumption that $s$ is a deadlock state, proving our claim by contradiction.

An obvious corollary is that for $M \sqsubseteq M'$ also hold: $M \models \delta \Rightarrow M' \models \delta$.

Parallel Composition and Refinement

One required property is that composition preserves refinement for the case of parallel composition. This will permit to later substitute refined automata for the more coarse abstraction when the required constraints are also respected.

**Lemma 3.** For any composable pair of automata $M_1$, $M_2$, and any automata $M_2'$ holds

$$M_2 \sqsubseteq M_2' \Rightarrow (M_1||M_2 \sqsubseteq M_1||M_2').$$

**Proof.** (sketch) For $M = M_1||M_2$ and $M' = M_1||M_2'$ we can from the relation $\Omega$ implied by the refinement $M_2 \sqsubseteq M_2'$ derive a relation $\Omega'$ required for the refinement $M \sqsubseteq M'$ $(M_1||M_2 \sqsubseteq M_1||M_2')$ as follows: For all $(s_1, s_2') \in S_1 \times S_2'$ and $(s_2, s_2') \in \Omega$ add $((s_1, s_2), (s_1, s_2'))$ to $\Omega'$. Due to the composition of $T$ resp. $T'$ from $T_1$ and $T_2$ resp. $T_2'$ we can easily prove condition 4 and 5.
3.2 Refinement

Parallel Composition and I/O-Refinement

Another required property is that composition preserves I/O-refinement for the case of parallel composition when no signals or labels are shared.

Lemma 4. For any composable pair of automata \( M_1, M_2 \), and any automaton \( M'_2 \) with \((I_2 - I'_2) \cap O_1 = \emptyset, (O_2 - O'_2) \cap I_1 = \emptyset, \) and \((\mathcal{L}(M_2) - \mathcal{L}(M'_2)) \cap \mathcal{L}(M_1) = \emptyset \) holds

\[
M_2 \sqsubseteq_{I/O} M'_2 \Rightarrow (M_1 \parallel M_2 \sqsubseteq_{I/O} M_1 \parallel M'_2).
\]

Proof. (sketch) For \( M' = M_1 \parallel M'_2 \) and \( M = (M_1 \parallel M_2)|_{I'/O'/P'} \) we are able to derive from the relation \( \Omega \) implied by the refinement \( M_2 \sqsubseteq_{I/O} M'_2 \) a relation \( \Omega' \) required for the refinement \( M \sqsubseteq_{I/O} M' \) as follows: For all \((s_1, s'_1) \in S_1 \times S'_2 \) and \((s_2, s'_2) \in \Omega \) add \(((s_1, s_2), (s_1, s'_2)) \) to \( \Omega' \). Due to the composition of \( T \) resp. \( T' \) from \( T_1 \) and \( T_2 \) resp. \( T'_2 \) we are able to prove condition 4. We can exploit our assumption that the additional signal sets and labels are disjoint and thus do not interfere. Therefore, condition 5 can be proven analogously. \(\Box\)

Parallel Composition and Restricted I/O-Refinement

For the restricted I/O refinement we can even replace the above assumption of disjoint signal sets and instead parallel composition can be preserved when the environment is complete and no deadlock in the refined composed system results.

Lemma 5. For any composable pair of automata \( M_1, M_2 \), and any automaton \( M'_2 \) holds

\[
M_2 \sqsubseteq^*_{I/O} M'_2 \Rightarrow \left( (M_1 \parallel M_2 \sqsubseteq_{I/O} M_1 \parallel M'_2) \vee (M_1 \parallel M_2 \models \delta) \right).
\]

Proof. (sketch) If \( M_1 \parallel M_2 \models \delta \) holds the right hand side of the equation becomes true anyway. Thus it remains to be proven that for \( M_1 \parallel M_2 \models \neg \delta \) from \( M_2 \sqsubseteq^*_{I/O} M'_2 \) follows \( M_1 \parallel M_2 \sqsubseteq_{I/O} M_1 \parallel M'_2 \).

For \( M' = M_1 \parallel M'_2 \) and \( M = (M_1 \parallel M_2)|_{I'/O'/P'} \) we can derive from the relation \( \Omega \) implied by the refinement \( M_2 \sqsubseteq^*_{I/O} M'_2 \) a relation \( \Omega' \) required for the refinement \( M \sqsubseteq_{I/O} M' \) as follows: For all \((s_1, s'_1) \in S_1 \times S'_2 \) and \((s_2, s'_2) \in \Omega \) add \(((s_1, s_2), (s_1, s'_2)) \) to \( \Omega' \). Due to the composition of \( T \) resp. \( T' \) from \( T_1 \) and \( T_2 \) resp. \( T'_2 \) we can prove condition 4. For condition 5 we can exploit our assumption that \( M_1 \parallel M_2 \models \neg \delta \) holds. As the restricted I/O-refinement ensures that only internal transitions of an internal state are modified, for any transition in \( M_1 \parallel M'_2 \) of a state \((s_1, s'_2) \in S_1 \times S'_2 \) which contradicts condition 5 there must exists a state \((s_1, s_2) \in S_1 \times S_2 \) with \( s_2 \) an internal state. We can conclude that the related state in \( (M_1 \parallel M_2)|_{I'/O'/P'} \) can only offer internal transitions or is a deadlock. While the later is excluded by the assumption \( M_1 \parallel M_2 \models \neg \delta \), in the former the condition 5 is obviously fulfilled. Therefore, I/O refinement between \( M_1 \parallel M_2 \) and \( M_1 \parallel M'_2 \) for \( M_1 \parallel M_2 \models \neg \delta \) have been proven. \(\Box\)
3.2.3 Ensuring Refinement

The defined notions for refinement differ w.r.t. the constraints and efforts which are required to ensure the preservation of refinement for the parallel compositions of automata.

We will first discuss how syntactical transformation rules and deadlock checks can be used to ensure correct refinement of one of the composed automata in Section 3.2.3 (a). In Section 3.2.3 (b) a general algorithm to check whether refinement holds for two automata is presented. Finally a procedure to synthesize the maximal automata w.r.t. refinement of a given automata and a constraint is presented in Section 3.2.3 (c).

(a) Syntactical Transformation Rules

We further distinguish two relevant cases. First, we consider a single automata and its refinement. Then, the case that within a parallel composition one automata should be refined without invalidating the beforehand valid refinement is considered.

(a-1) The Refinement of a Single Automaton

For the case of a given abstract behavior in form of an automata, the required correct more detailed behavior can be systematically derived using a number of valid transformations. Examples for such valid transformation steps for refinement are:

- If from one state more than one transition with the same I/O is present, one edge can be erased.

- For each edge leading to a target state $s$ an edge with identical I/O and target state $s'$ can be added, as long as all edges leading from $s$ to another state $s''$ is copied such that an edge with identical I/O leads from $s'$ to $s''$.

While this minimal set of transformation rules is rather restricted, the combination of these steps permits to build arbitrarily complex correct refinements of the original automaton.

For I/O refinement we get a number of additional valid transformation steps which further permit to adjust the I/O signal sets as long as they do not change w.r.t. the I/O signal sets of the refined automata.

For restricted I/O refinement the additional valid transformation steps are further restricted as they only permit to extend transitions of internal states. The additional transformation step is:

- A transition of an internal state can be replaced by a transition with additional I/O.

In Figure 1 A more complex transformation scheme derived from the rules above is presented. It is to be noted, that it describes the transformation of statecharts rather than real-time automata and thus the edge labelling $e_i/e_j$ will denote in the related real-time automata model two transitions labelled with input/output set pair ($\{e_i\}, \emptyset$) and input/output set pair ($\emptyset, \{e_j\}$). We use $e_i$ to denote externally visible signals relevant for the coordination.

It assumes that we have an initial edge leading from state $S_1$ to any of the states of the arbitrary subgraph visualized as a cloud. The states within the cloud can be only connected by
3.2 Refinement

internal transitions and the resulting automaton will non-deterministically choose an internal transitions before one of the transitions to \(s_2, \ldots, s_n\) with resp. send event \(e_2, \ldots, e_n\) is chosen. Such a local operating subgraph can be replaced by any arbitrary subgraph which may interact with the context by sending and receiving any internal signals denoted by \(\square\) as long as it guarantees that for each path through the new subgraph to one of the states \(s_2, \ldots, s_n\) a path with similar delay exists in the original cloud which leads to the same state.

(a-2) Refinement of a Parallel Composition of Automata

In the case that the parallel composition of a set of automata should be refined, while ensuring refinement w.r.t. one of them, valid transformation rules have to ensure also that deadlock freedom is preserved. This is a rather hard problem as we have to take into account the overall behavior of all automata. We are however able to avoid this problem by assuming that the resulting parallel composition will be checked for deadlocks. Thus we require only a transformation that ensures refinement if no deadlock occurs. For restricted I/O refinement, as outlined in Lemma 5, a deadlock check is sufficient to ensure that I/O refinement holds for a parallel composition.

The in the transformation scheme of Figure 1 added signals can be employed to coordinate the required or possible decisions as required. We can proof that the described transformation does indeed ensure refinement as follows. The above transformation scheme ensure that \(M'_1 \sqsubseteq^*_{I/O} M_1\) holds. If \(M'_1 \parallel M_2 \models \neg \delta\) has been checked, we can conclude using Lemma 5 that \(M'_1 \parallel M_2 \sqsubseteq_{I/O} M_1 \parallel M_2\) holds. If also \(L(M_2) \cap L(M_1) = \emptyset\), \(I_2 \cap O_1' = \emptyset\), and \(I'_1 \cap O_2 = \emptyset\) holds, we have due to Lemma 4 also \(M'_1 \parallel M_2 \sqsubseteq_{I/O} M_1\).

(b) Checking Refinement

To check for two automata \(M\) and \(M'\) whether a refinement relation exists can be done systematically using the following algorithm.

We start the processing simply with \(S_0 = Q \times Q'\). Then, we compute first all reachable state combinations as the fix-point of:

\[
S_{i+1} = \{(s_t, s'_t) \in S_i | (s_s, s'_s) \in S_i \land (s_s, A, B, s_t) \in T \land (s'_s, A, B, s'_t) \in T'\},
\]

where \(S_{k+1} = S_k\) holds. Then, we set \(\Omega_0 = S_k\) and compute the largest fix-point for condition 4 and condition 5 with the following two steps:

\[
\Omega_{i+1}^{tmp} = \{(s_{1}, s'_{1}) \in \Omega_i | \forall (s_1, A, B, s_2) \in T \exists (s'_1, A, B, s'_2) \in T' : (s_2, s'_2) \in \Omega_i\}
\]

\[
\Omega_{i+1} = \{(s_{1}, s'_{1}) \in \Omega_{i+1}^{tmp} | \forall (s'_1, A', B', s'_3) \in T' : \exists (s_1, A', B', s_3) \in T\}
\]
When a fix-point with $\Omega_{i+1} = \Omega_i$ is reached, the maximal refinement relation $\Omega_i$ has been computed. Thus if for both automata $M$ and $M'$ hold $M \subseteq M'$ the above algorithm computes a correct refinement relation which contains for every $s \in Q$ at least one pair in $\Omega \cap (Q \times Q')$. This algorithm can be straightforward adjusted to also check I/O refinement as well as restricted I/O refinement.

(c) Synthesis of Maximal Refining Automata

The refinement notion results in a partial ordering on the set of all automata. Therefore, w.r.t. a specific invariant $\psi$ we are able to derive the maximal refined automaton $M'$ that satisfies $\psi$ and also refines a given automaton $M$ as follows:

We start the processing simply with $S_0 = \{s \in S | M, s \models \psi\}$. Then, we compute the largest fix-point for a property $\psi$ and condition 5 with the following step:

$$S_{i+1} = \{s_1 \in S_i | \forall (s_1, A, B, s_2) \in T \ \exists (s_1, A, B, s_2') \in T : s_2' \in S_i\}$$

When a fix-point with $S_{i+1} = S_i$ is reached, a maximal automaton $M'$ which refines $M$ has been computed. For the automaton $M'$ further holds $M' \models \psi$.

It is to be noted that the proof that the above algorithm does indeed compute the maximal refinement requires that the implicitly by the above equation defined function $(f_{\models, \psi}(S)) := \{s_1 \in S | \forall (s_1, A, B, s_2) \in T \ \exists (s_1, A, B, s_2') \in T : s_2' \in S_i\}$ on the state set is monotonous and determines uniquely which states have to be erased. While this holds for the defined notion of refinement and invariants, in the general case of compositional constraints, ACCTL or even CTL formulas this is not the case.

3.3 Compositional Reasoning

This subsection formally underpins the later employed compositional verification approach. The approach yields a verification result for the overall system without building its complete state space.

3.3.1 Definitions

For our approach the interesting class of constraints are those, which are preserved under refinement and composition with disjoint labelling.

**Definition 16.** A constraint $\phi$ is compositional iff for any automata $M_1$, $M'_1$, and $M_2$ with $L(M_2) \cap L(\phi) = \emptyset$ holds

$$ (M_1 \models \phi) \Rightarrow ((M_1 \parallel M_2 \models \phi) \lor (M_1 \parallel M_2 \models \delta)) \quad \text{and} \quad (M_1 \subseteq M'_1) \land (M'_1 \models \phi) \Rightarrow (M_1 \models \phi) \quad (9)$$

$$ (M_1 \parallel M_2 \models \phi) \lor (M_1 \parallel M_2 \models \delta) \quad (10)$$

Note that due to incompatible overlapping labelling functions the parallel composition of two automata might be empty. As we consider such an empty automaton to have a deadlock the above definition also includes this case.
3.3 Compositional Reasoning

3.3.2 Characteristics

For invariants we are first able to show compositionallity and thus preservation for a composition with an automaton with disjoint labelling.

**Lemma 6.** Invariants are compositional constraints.

*Proof.* If $M_1 \parallel M_2$ is empty we have $M_1 \parallel M_2 \models \delta$. Therefore, it remains to show that for two automata $M_1$ and $M_2$ and an invariant $\psi$ with $\mathcal{L}(\psi) \cap \mathcal{L}(M_2) = \emptyset$ holds $(M_1 \models \psi) \Rightarrow (M_1 \parallel M_2 \models \psi)$ to proof condition 9. For $M = M_1 \parallel M_2$ all states $(s_1, s_2) \in S$ hold $s_1 \in S_1$. From $M_1 \models \psi$ follows for all $s_1$ that $M_1, s_1 \models \psi$ holds. Due to $\mathcal{L}(\psi) \cap \mathcal{L}(M_2) = \emptyset$ we can conclude that no $s_2 \in S_2$ invalidates $\psi$. Combining both facts we have for all $(s_1, s_2) \in S_1 \times S_2$ that $M_1 \parallel M_2, (s_1, s_2) \models \psi$ holds and we thus have $M_1 \parallel M_2 \models \psi$. As any refinement only erases states condition 10 follows also.

A time-bound restricts the upper and/or lower number of time steps allowed in between two states. E.g., denoting that between message send and delivery at least $x$ and at most $y$ time steps may occur. Thus a time-bound is actually an upper and/or lower bound on the length of all paths between two states. For example, the according ACCTL formula for a trigger condition $p_1$ and a condition $p_2$ which has to be reached within the lower and upper bound of steps would thus be: $\text{AG}(\neg p_1 \lor (\text{AF}_{[x,y]} p_2))$ (which is equal to the better comprehensible form $\text{AG}(p_1 \Rightarrow (\text{AF}_{[x,y]} p_2))$). This kind of constraint can also be proven to be compositionallity.

**Lemma 7.** Time-bounds are compositional constraints.

*Proof.* We can prove for any time-bound $\phi$ that for two automata $M_1$ and $M_2$ condition 9 ($(M_1 \models \phi) \Rightarrow ((M_1 \parallel M_2 \models \phi) \lor (M_1 \parallel M_2 \models \delta))$ holds by using the fact that the parallel composition would at most result in erasing a number of transitions and states. Thus, no longer or shorter paths can result. The same argument holds for refinements and thus condition 10 also follows.

In general for ACTL formulas or its clocked version ACCTL we can prove compositionallity.

**Lemma 8.** All ACCTL formulas are compositional constraints.

*Proof.* By induction over the formulas we are able to proof our claim as follows: All propositional formulas are invariants and thus due to Lemma 6 are compositional. For a compositional ACCTL formula $\phi$ and $\psi$ also $\text{AX}_{[a]} \phi$, $\text{AF}_{[a,b]} \phi$, $\text{AG}_{[a,b]} \phi$, $\text{A}(\phi \ U_{[a,b]} \psi)$, and $\text{A}(\phi \ R_{[a,b]} \psi)$ are compositional, because $\phi$ or $\psi$ holds that the refinement or composition with disjoint labelling sets only reduces the possible set of relevant runs. As the formula also holds when the set of relevant runs becomes empty, in any case these properties are preserved when $\phi$ and $\psi$ are preserved. Thus by induction we have proven that all ACCTL formulas are compositional.

Invariants, upper and lower time-bounds, and ACCTL formulas in general are constraints which refer only to all possible paths. Thus using the fact that a refinement or composition with disjoint labelling sets only reduces the possible sequences of states with identical labelling, they have been proven to be compositional. That deadlock freedom is also compositional follows by construction for condition 9 and Lemma 2 for condition 10 as follows:
Lemma 9. Deadlock freedom is a compositional constraint.

Proof. To show condition 9 we have to prove it for $\phi = \neg \delta$ which results in $(M_1 \models \neg \delta) \Rightarrow ((M_1 \parallel M_2 \models \neg \delta) \lor (M_1 \parallel M_2 \models \delta))$. This holds as $M_1 \parallel M_2 \models \neg \delta \lor \delta$ is always true. Due to Lemma 2 condition 10 also follows.

Compositionallity has thus been established for most of the properties required for real-time systems such as deadlock freedom, upper bounds for the maximal delays of message transports, lower bounds for the minimal delays of message transports, and invariants.

In contrast to this result, general temporal logic formulas which state explicitly that a specific has to be eventually reached (abstracting from possible effects of non-determinism) may not be preserved.\(^3\)

Properties of Composed Systems

For a substitution of an I/O refinement that only adds disjoint I/O signals and labels, we further have to proof that compositional constraints and deadlock freedom are preserved.

Lemma 10. For automata $M_1$, $M_2$, and $M_2'$ with $M_2 \sqsubseteq_{I/O} M_2'$ with $I_1 \cap (O_2 - O_2') = \emptyset$, and $O_1 \cap (I_2 - I_2') = \emptyset$, and $L(M_1) \cap (L(M_2) - L(M_2')) = \emptyset$ and any compositional constraint $\phi$ holds

\[(M_1 \parallel M_2' \models \phi \land \neg \delta) \Rightarrow (M_1 \parallel M_2 \models \phi \land \neg \delta)\]  \hspace{1cm} (11)

Proof. Due to $\phi$ and $\neg \delta$ being compositional and Definition 16 we can for $M_2'' = M_2|_{I_2/O_2'/P_2'}$ conclude that $M_1 \parallel M_2'' \models \phi \land \neg \delta$ or $M_1 \parallel M_2'' \models \delta$. Due to Lemma 2 and 3 we even have $M_1 \parallel M_2'' \models \phi \land \neg \delta$. From $I_1 \cap (O_2 - O_2') = \emptyset$, $O_1 \cap (I_2 - I_2') = \emptyset$, and $L(M_1) \cap (L(M_2) - L(M_2')) = \emptyset$ follows that $M_2$ adds to $M_2''$ only I/O and labels that do not interfere with $M_1$ and thus $M_1 \parallel M_2$ has the same reachable state set and transitions and thus $M_1 \parallel M_2 \models \neg \delta$. As $\phi$ is only interpreted over states and the labelling is identical, also $\phi$ must hold and thus condition 11 is proven. \(\blacksquare\)

Using the calculus and lemmata presented in this section, we are able to verify pattern-based real-time systems as outlined in the following section.

4 Pattern-Based Real-Time System Modeling

We start our presentation of the approach for correct pattern-based design and verification with the definition of the main elements such as patterns, components and the employed notion of a system in Section 4.1. Then, in Section 4.2 we present the main compositionallity result and discuss its benefits. The implied process is then outlined in Section 4.4 after reviewing the automated support for different design and verification tasks within the approach in Section 4.3.

\(^3\)Noted that required behavior can be easily designed into the automata itself. The refinement will ensure that only explicitly via non-determinism under-specified behavior can be resolved while all deterministically specified aspects of the behavior will be preserved. E.g., when all requests are answered within a non-deterministic protocol within a given time frame this will also hold for any of its refinements.
4.1 Patterns, Components and System

In this section we outline how a system can be constructed by means of pattern and components. In the following we define the employed notions for patterns, components, and the resulting system notion.

4.1.1 Pattern Definition

In our approach a pattern comprises of a set of roles that interact only via a connector that will connect the related component ports in the final system. We further have the restriction that for each pattern we have to specify a protocol automaton and invariants for each role. An overall constraint in form of a CCTL formula is also possible. While usually in untimed models the connector behavior is omitted, channel delay and reliability are of crucial importance for real-time systems and thus have to be addressed explicitly in form of an additional connector automaton.

A pattern is formally defined as follows:

**Definition 17.** A pattern $P$ is a 4-tuple $(M, \Psi, \phi, M_N)$ with a set $M$ of automata $M_1, \ldots, M_k$ for each role, a set $\Psi$ of invariants $\psi_1, \ldots, \psi_k$ for each role, the pattern constraint $\phi$, and the connector automaton $M_N$.

It is to be noted, that the classical notion of an interface is also included in this definition of a pattern. We simply restrict the pattern to only two roles which are only allowed to exchange the messages defined within the interface. In contrast to classical syntactical interface notion, however, the roles require that each interface is equipped with a real-time protocol describing its quality of service characteristics. The explicit consideration of a connector further ensures that besides the client and server protocol also the effects of the communication medium are taken into account.

Syntactical correct pattern are restricted to such ones where the pattern constraints $\phi$ are compositional. A proper labelling has further to ensure for all pattern roles that $\mathcal{L}(M_i) \cap \mathcal{L}(M_j) = \emptyset$ for all $i \neq j$.

To ensure semantical correctness for a pattern, we verify whether the behavioral requirement specified by means of a compositional CCTL formula hold for a pattern. If the requirement holds, the pattern is named *correct*.

**Definition 18.** A pattern $P = (M, \Psi, \phi, M_N)$ with a set $M$ of automata $M_1, \ldots, M_k$ is locally correct iff:

$$M_1 \parallel \ldots \parallel M_k \parallel M_N \models \phi \land \neg \delta$$  \hspace{1cm} (12)

This can be verified using a real-time model checker. First we have to build the model $M_1 \parallel \ldots \parallel M_k \parallel M_N$ and then have to check whether the constraint $\phi \land \neg \delta$ holds.

For proving the correctness of all $n$ patterns ($n'$ different ones) of a system, we will have $n'$ checks in $O(\exp(k))$, where $k$ is the maximal number of roles per pattern. Domain specific restrictions and good software design usually guarantee a fixed upper bound for $k$ for arbitrary $n$, because the number of roles per pattern will not further increase when more components and
patterns are added to the system. Thus, the required verification becomes possible when the state space of each single pattern is not too large.

Pattern with \(k_i\) roles and one connector

Overall behavior:
\[ M^P_i = M^P_{i,1} \parallel \ldots \parallel M^P_{i,k} \parallel M^P_{i,N} \]

Correctness:
\[ M^P_i \models \phi_i \land \neg \delta \]

Figure 2: Definitions for a locally correct pattern

For multiple patterns \(P_1, \ldots, P_n\) we refer to their constraint as \(\phi^P_i\), the connector automaton as \(M^P_{i,N}\), and the role protocols as \(M^P_{i,1}, \ldots, M^P_{i,k}\). The overall behavior \(M^P_i\) relate to the pattern can thus be build by \(M^P_{i,1} \parallel \ldots \parallel M^P_{i,k} \parallel M^P_{i,N}\). The related notations and required condition for local correctness are summarized in Figure 2.

If for a resulting system \(S\) with behavior \(M^S\) holds \(M^S \models \phi_i \land \neg \delta\) we say that the system preserves the correctness of the pattern \(P_i\).

4.1.2 Component Definition

Common understanding is that a component only interacts with its environment via well-defined interfaces (cf. [Szy98]). The interfaces of components in the presented approach are denoted by the related pattern roles and their protocols and invariants. To serve and use its interfaces in a correct manner, a component design has to coordinate and refining each role automaton. Based on Definition 13 such a refinement has to respect the role automaton (do not add possible behavior or block guaranteed behavior) and additionally has to respect the guaranteed behavior of the roles in form of its invariants.

Formally, we can thus define a component as follows:

**Definition 19.** A component \(C\) is a triple \((\mathcal{M}, \Psi, M^C)\) with a set \(\mathcal{M}\) of automata \(M_1, \ldots, M_h\) the realized pattern roles for each port, the set \(\Psi\) of all associated role invariants \(\psi_1, \ldots, \psi_h\), and the overall component behavior in form of a single automaton \(M^C\).

The component role invariant \(\psi^C\) can be derived by simply combining the related role invariants \((\psi_1 \wedge \cdots \wedge \psi_h)\) of \(\Psi\). Additional local constraints may be added as long as \(\psi^C\) remains a compositional property.

For a proper labelling of the pattern role automata of the components we expect \(\mathcal{L}(M^C_i) \cap \mathcal{L}(M^C_j) = \emptyset\) for any \(i \neq j\). By choosing appropriate labelling functions for the overall component automaton \((\mathcal{L}(M^C_i) \supseteq \mathcal{L}(\psi^C_i))\) we can achieve that accordingly adjusted combinations
of the invariants of all port roles ensure that each component remains in a proper state w.r.t. the requirements of its patterns.

Besides the patterns also the components have to be verified. We therefore have to verify that the role invariants hold for the component behavior and that it also respects each role automaton denoted by the following notion of a correct component.

Definition 20. A component $C = (\mathcal{M}, \Psi, M^C)$ with a set $\mathcal{M}$ of automata $M_1, \ldots, M_h$ is a locally correct iff it holds:

$$M^C \sqsubseteq_{I/O} M_1 \parallel \ldots \parallel M_h \quad \text{and} \quad M^C \models \psi^C \land \neg \delta$$ \hspace{1cm} (13)

We can again use a real-time model checker to prove $\psi^C \land \neg \delta$ for $M^C$. To ensure that $M^C$ refines each of the role protocols associated to its ports we can either use the refinement check sketched in Section 3.2.3 (b) or the syntactical refinement rules for restricted I/O refinement and checking for deadlocks might be employed (cf. Section 4.3.2).

Proving the correctness of all $m$ components ($m'$ different ones) of a system requires $m'$ checks in $O(\exp(h))$, where $h$ is the maximal number of roles per component. Like in the case of patterns, usually a fixed upper bound for $h$ can be assumed, as in most domains and for good software designs hold that the number of ports per component will not further increase when more components and patterns are added to a system.

For multiple components $C_1, \ldots, C_m$ we refer to their overall component behavior as $M^C_j$ and invariant as $\psi^C_j$. The port protocols are denoted as $M^C_{j,1}, \ldots, M^C_{j,h_j}$. For a summary of the definitions for a locally correct component see Figure 3.

If for a resulting system $S$ with behavior $M^S$ holds $M^S \models \phi^C_j \land \psi^C_j \land \neg \delta$ we say that the system preserves the correctness of the component $C_j$.

4.1.3 System Definition

Our approach assumes that the required system can be built by a number of components and patterns which overlap at their ports resp. roles. This does not restrict our approach, because the employed notion of a pattern with roles includes also more traditional interface definitions as long as they include the interface protocol and quality of service attributes. To instantiate a
component as well a pattern the employed I/O signals have to be relabelled in such a manner that connected roles, connectors and components are relabelled in a consistent manner.

Therefore, we assume that instead of the original pattern or component definitions an appropriate relabelling (cf. Definition 9) has been employed that ensures that all pattern and component instances have their own disjoint I/O signal and labelling sets. The component instances have to be adjusted accordingly to ensure that they can interact with the I/O signals as specified by the related pattern roles. The same holds for the atomic propositions, which have to be adjusted such that all components have a disjoint labelling. Within a single component, in contrast, the same label must be employed by different role constraints to ensure that conflicts between the different roles can be safely detected.

Using these assumptions, a system can be formally defined as follows:

**Definition 21.** A system \( S \) is a triple \( (P, C, \text{map}) \) with a set \( P \) of pattern instances \( P_1, \ldots, P_n \), a set \( C \) of component instances \( C_1, \ldots, C_m \), and a mapping \( \text{map} \) which assigns to each component instance port the related unique pattern instance role.

The syntactical correctness of such a system composed of pattern and component instances requires that all related automata \( M_{P_i}^{P_1}, \ldots, M_{P_n}^{P_n}, M_{C_j}^{C_1}, \ldots, M_{C_m}^{C_m} \) are connected accordingly by \( \text{map} \) such that all roles are realized by the component ports.

**Definition 22.** A system \( S = (P, C, \text{map}) \) is syntactical correct iff \( \text{map} \) is a partial bijective mapping between the set of all ports of any component instance in \( C \) and the set of all pattern instance roles of \( P \). The mapping must further ensure that the pattern role and port protocols are equivalent (bisimilar):

\[
\forall M_{i,k}^{P_i} = \text{map}(M_{j,h}^{C_j}) : M_{i,k}^{P_i} \cong M_{j,h}^{C_j}
\]

If \( \text{map} \) is total we have a closed system where each component instance port is uniquely mapped to a compatible pattern instance role.

To describe next what is actually verified compositionally within our approach, we first have to define what it means that the system is semantically correct.

**Definition 23.** For \( S = (P, C, \text{map}) \) with a set \( P \) of pattern instances \( P_1, \ldots, P_n \), a set \( C \) of component instances \( C_1, \ldots, C_m \), and a bijective mapping \( \text{map} \) semantical correctness holds iff the pattern constraints \( \phi_i^P \) and component invariants \( \psi_j^C \) also hold for the system itself:

\[
M_{1}^{P, N} \parallel \ldots \parallel M_{n}^{P, N} \parallel M_{1}^{C} \parallel \ldots \parallel M_{m}^{C} \models \phi_1^P \land \cdots \land \phi_n^P \land \neg \delta \quad \text{(14)}
\]

\[
M_{1}^{P, N} \parallel \ldots \parallel M_{n}^{P, N} \parallel M_{1}^{C} \parallel \ldots \parallel M_{m}^{C} \models \psi_1^C \land \cdots \land \psi_m^C. \quad \text{(15)}
\]

Common modular approaches result in a disjoint decomposition of the system. In our approach, however, we have overlapping models where the specified role protocols of each pattern and parallel operating protocol refinement of the components refer to the same port. These sets of ports and roles are employed as maximal non-deterministic context for the components as well as guaranteed behavior of each pattern role.
4.2 Compositional Reasoning

As depicted in Figure 4, the overlapping border between the basic elements (patterns and components) are always well-defined protocols both sides agreed upon. The real-time character of the protocols ensures that unrestricted blocking effects are excluded. Thus, deadlock freedom can be proven compositionally only by referring to the independent composition of all port protocols. It is to be noted that in non-timed models a similar approach will not be possible, as worst-case blocking times are not explicitly considered and therefore cyclic blocking effects have to be taken into account (see [Gie00]).

4.2 Compositional Reasoning

Due to the compositional nature of our approach, an additional step to perform verification for the overall system to prove semantical correctness after its composition is not required. We will show that only a proof for syntactical correctness is required to derive the semantical correctness. In the remainder of this subsection, we argue why semantical correctness can be achieved via syntactical correctness only.

In a first part in Section 4.2.1 we demonstrate how local properties of patterns and components are preserved by composition. Then, in Section 4.2.2 we show how non-local properties can also be checked. An extension of the approach towards incremental model checking is then reviewed in Section 4.2.3. Finally, we discuss the impact of local changes on the amount of verification which is required to again establish the beforehand proven properties in Section 4.2.4.

4.2.1 Local Properties

For the restricted class of compositional properties (see Definition 16) we can also use the border built by the ports resp. roles to proof also the required pattern properties $\phi_i^P$ and component properties $\psi_j^C$ compositionally. The fundamental result is that a closed system with only correct pattern instances and only correct component instances is semantically correct if all elements are syntactically correct connected.

**Theorem 1.** A syntactically correct closed system $S = (P, C, map)$ with a set $P$ of correct pattern instances $P_1, \ldots, P_n$ and a set $C$ of correct component instances $C_1, \ldots, C_m$ is semantically correct.

**Proof.** An overview about the proof idea is depicted in Figure 5. First in the initial step of Figure 5, for any $i \in [1, n]$ we can conclude for each correct pattern $P_i = (\{M_1, \ldots, M_k\}, \Psi_i, \phi_i, M_i^P)$...
that it fulfills its constraint \( \phi_i \) and is deadlock free: \( M_i^P = M_i^P \parallel \ldots \parallel M_{i,k}^P \parallel M_i^{P,N} \models \phi_i \land \neg \delta \). As all \( M_i^P \) have disjoint signal and labelling sets (\( L(M_i^P) \cap L(M_j) = \emptyset \) for \( i \neq j \)), we can combine them in the step (2) of Figure 5 and have: \( M_i^P \parallel \ldots \parallel M_n^P \models \phi_1 \land \ldots \land \phi_n \land \neg \delta \). We can then in step (3) of Figure 5 rearrange the different role protocols \( M_{i,k}^P \) such that they ordered w.r.t. their correspondence to the components using the bisimilar port automata \( M_j^{C,k} \).

For a locally correct component instance \( C_j \) with behavior \( M_j^C \) and related pattern role protocols \( M_{j,1}^C, \ldots, M_{j,h_j}^C \) holds also \( M_j^C \sqsubseteq_{I/O} M_j^C \parallel \ldots \parallel M_{j,h_j}^C \). We thus can due to Lemma 10 conclude that \( \phi_1 \land \ldots \land \phi_n \land \neg \delta \) also holds for \( M_1^{P,N} \parallel \ldots \parallel M_n^{P,N} \parallel M_1^C \parallel \ldots \parallel M_m^C \) which results when \( M_{j,1}^C \parallel \ldots \parallel M_{j,h_j}^C \) is replaced by \( M_j^C \). Condition 14 for a semantically correct system has thus been proven.

To proof condition 15 we can simply derive from the fact that \( M_i^C \) a correct component for any \( i \) that \( M_i^C \models \psi_i^C \) must be fulfilled. As \( \psi_i^C \) are restricted to be compositional, we have due to condition 5 of Definition 13 that \( M_1^{P,N} \parallel \ldots \parallel M_n^{P,N} \parallel M_1^C \parallel \ldots \parallel M_m^C \models \psi_i^C \) holds and the above result of deadlock freedom. By iteration over all \( i \in [1,m] \) we thus can also obtain condition 15.

Therefore, we can conclude that the pattern constraints as well as each role invariant also hold for the resulting composed system. It is to be noted that instead of invariants \( \psi \) also compositional temporal logic formulas might be employed to restrict the component behavior. In our experiments so far, however, invariants have been sufficient, because dynamic issues are better addressed in an explicit fashion using the protocol automata.

The advantage of the compositional approach is that Theorem 1 permits us to verify condition 14 and 15 without building the state space for \( M_1^{P,N} \parallel \ldots \parallel M_n^{P,N} \parallel M_1^C \parallel \ldots \parallel M_m^C \). Instead, only the syntactical correctness of the overall system and correctness for all patterns and components has to be checked. The parallel composition of all components and patterns in the overall system can result in one check in \( O(\exp(n + m)) \) due to the possibly exponential growing product

![Figure 5: Sketch of the proof idea](image-url)

\[ \text{that it fulfills its constraint } \phi_i \text{ and is deadlock free: } M_i^P = M_i^P \parallel \ldots \parallel M_{i,k}^P \parallel M_i^{P,N} \models \phi_i \land \neg \delta. \]
state space of the system. For \( n' \) the number of different patterns and \( m' \) the number of different components the sum of all checks for our approach is in \( O(n' \exp(h) + m' \exp(k)) \) for \( k \) the fixed maximal number of roles per pattern and \( h \) the fixed maximal number of roles per component. The efforts required to check the system do therefore only grow linear with the number of different patterns and components which make the approach scalable. The compositional nature even permits to restrict the required checks to such parts which are changed or added as further outlined in Section 4.2.4.

We only require that \( k \) and \( h \) are not too large constants such that the required local checks remain feasible. Then, we can derive in our example the correct operation for any arbitrary large finite set of shuttle components which are correctly interconnected via the patterns for distance control.

### 4.2.2 Non Local Properties

The presented approach results in the restriction that only local properties for each pattern or component can be proven. When also the verification of properties \( \phi_s \) which involve more than one component is required we have to slightly adjust our approach.

A standard approach to compositional reasoning is to use manually derived local properties which combination ensures that the required overall property can be derived for the overall system. In contrast to this standard assume/guarantee approach, in the presented approach we do not have to develop an appropriate decomposition of the required property. We can, if the local checks are not sufficient, exploit the fact that proper subsets of the interconnected components and patterns can be seen as components and patterns again.

### Reasoning via Composed Components

A first, rather trivial case is that a set of connected components and patterns with only unconnected component ports again form a composed component (see Figure 6).

![Figure 6: Example of a composed component](image)

Therefore, we can instead of each contained component verify \( \phi_s \) for the composed system of components and ensure that \( \phi_s \) also holds for the overall system using Theorem 1 with that composed component.

**Definition 24.** A composed component \( U \) is a 5-tuple \((\mathcal{P}, \mathcal{C}, \text{map}, \mathcal{R}, \phi_s)\) with \( \mathcal{P} \) a set of patterns \( P_1, \ldots, P_n \) and \( \mathcal{C} \) a set of components \( C_1, \ldots, C_m \) with syntactically correct connected...
pattern roles and component ports (map), and \( \mathcal{R} \) the set of unconnected component protocols \( M_1, \ldots, M_h \).

We can build the related component \( C = (\mathcal{M}, \Psi, M^s) \) for the composed component \( U \) as follows:

- \( \mathcal{M} = \{M_1, \ldots, M_h\} \) is the set of unconnected port protocols,
- \( \Psi \) is the set of related invariants \( \psi_1, \ldots, \psi_k \), and
- \( M^C \) is the component automaton build by \( M_{1P,N}^p \| \cdots \| M_{nP,N}^p \| M_{1C}^c \| \cdots \| M_{mC}^c \).

Properties defined across the patterns and contained components can thus be verified in a compositional manner for any system containing the composed component using Theorem 1. We only have to add the non-local compositional property \( \phi_s \) to \( \psi^C \) accordingly. The embedding into any syntactical correct context will then preserve the included non-local compositional property \( \phi_s \).

A second, more unexpected case is a subset of the system with only unconnected pattern roles, which results in a subsystem which w.r.t. the presented verification approach can be addressed like a pattern (see Figure 7).

**Figure 7: Example of a pattern web**

We can first verify a property \( \phi_s \) for this web of patterns which also includes components locally. Using Theorem 1 we can then conclude that \( \phi_s \) also holds for the overall system.

**Definition 25.** A pattern web \( W = (\mathcal{P}, \mathcal{C}, \text{map}, \mathcal{R}, \phi_s) \) with \( \mathcal{P} \) a set of patterns \( P_1, \ldots, P_n \) and \( \mathcal{C} \) a set of components \( C_1, \ldots, C_m \) with syntactically correct connected pattern roles and component ports (map), and \( \mathcal{R} \) the set of unconnected pattern roles \( M_1, \ldots, M_k \).

For such a pattern web \( W \) we can build the related composed pattern by \( P = (\mathcal{M}, \Psi, \phi, M^P) \) as follows:

- \( \mathcal{M} = \{M_1, \ldots, M_h\} \) is the set of role protocols,
- \( \Psi \) is the set of related invariants \( \psi_1, \ldots, \psi_k \),
- \( \phi \) is the required cross component property \( \phi_s \), and
- \( M^P \) is the connector automaton build by \( M_{1P,N}^p \| \cdots \| M_{nP,N}^p \| M_{1C}^c \| \cdots \| M_{mC}^c \).
Therefore, also cross component properties can be verified in the compositional manner using Theorem 1. We only have to consider that pattern web which contains a proper subset of the overall system which contains all components referenced in the property $\phi_s$.

In both cases the state space we have to build to model check $\phi_s$ will for appropriate connection structures still avoid the exponential blow up.

4.2.3 Incremental Checking

It is to be noted, that for pattern webs, composed components, or the case of only checking the basic components and pattern depending on the required property a check can result in a false negative. This may happen when in the specific environment the detected counter examples are not possible, while the more general environment built by the protocols includes them. Thus also considering the behavior of a set of additional elements may be required to prove the property when these patterns and components actually contributes to the required goal.

A first rather trivial case is that a set of connected components and patterns with only unconnected component ports again form a composed component.

Therefore, a possible strategy is to extend the considered subsystem in form of a composed component or pattern web as long as the property cannot be checked and the detected counter examples may be excluded by the environment. We can do this by simply extend the considered pattern web or composed component with other interacting elements as long as the property cannot hold even in a more restricted environment (see $M_1, M_2, \ldots$ in Figure 8).

However, in a good software design and architecture behavioral dependencies between different parts of a system should be made explicit to exclude problems when changing the system. If such information is available, the proposed concepts for verification of pattern web or composed component properties allow to safely redesign each pattern web or composed component. If in contrast a large fraction of the system is required to prove a certain property, the decoupling between the elements of the subsystem is rather weak.
4.2.4 Checking and Changes

In practice, no system is build from scratch in an straight forward sequence of correct modeling steps. Instead, in an incremental process the different subsystems, patterns, and components will be designed, checked, redesigned and so on. Therefore, the impact of changes and the portion of the system which has to be reevaluated resp. verified for each change is crucial importance.

In the case of only local properties, we only have to redo the verification steps which refer to the changed elements. As such changes do not invalidating any other earlier done verification steps for other elements checking the modified elements is enough.

If also non-local properties have been checked using a pattern web or composed component, these checks have to be repeated if one in the check employed element has been changed. The same holds for checks which are based on the incrementally extension of the checked subsystem. Therefore, it is in both cases useful to store which pattern web or composed component has been finally used to proof a specific property.

In any of the cases, if the changes have resulted in a additional dependency to an already contained element or added one, the check might require to integrate also this element into the checked fraction of the system.

4.3 Design and Verification Steps

Based on the definitions of the preceding sections, a number of possible design and verification steps can be employed.

First, we briefly review options to support the design of patterns in Section 4.3.1. Then, in Section 4.3.2 two more concrete scenarios for the support of the component design are discussed.

4.3.1 Pattern Design

For the pattern design itself no specific support has been proposed within the presented approach. Possible options for further automation are here scenario-based synthesis [SKK01] or protocol synthesis [GvB90, Sal96]. In [Gie03] a first approach towards the scenario-based synthesis of patterns and their role protocols can be found. If a global specification of the pattern behavior is available, protocol synthesis is however the better choice.

4.3.2 Component Design

For the design of components in contrast at least three different options result from the outlined approach: (a) restricted I/O refinement for the port protocols and local checks for deadlock freedom can ensure correct component behavior, (b) correct component behavior can be synthesized such that the resulting behavior is the maximal refinement for a given constraint and the pattern role invariants, and (c) the component behavior can be realized manually and an explicit check for I/O refinement has to be done. As case (c) is straight forward, we review in the following only case (a) of a port refinement and explain which syntactical transformation steps and verification steps are required for it and give a short overview about case (b).
(a) Port Refinement

The restricted I/O refinement for refined versions of the port protocols of a component and a local check for deadlock freedom can ensure that the component behavior is indeed a correct refinement of its port protocols. To refine the pattern role protocols of each component port in isolation, we additionally require an internal automata which coordinated them.

These ideas can formally defined as a via port refinement derived component (in contrast to Definition 19) as follows:

**Definition 26.** A via port refinement derived component $C$ is a 4-tuple $(M, \Psi, M^r, M^s)$ with a set $M$ of automata $M_1, \ldots, M_h$ of the realized pattern roles, the set $\Psi$ of all associated role invariants $\psi_1, \ldots, \psi_h$, a set $M^r$ of automata $M^r_1, \ldots, M^r_h$ refining the realized pattern roles, and the component internal synchronization automaton $M^s$.

The overall component automaton $M^C$ for such a via port refinement derived component is built by $M^r_1 \parallel \ldots \parallel M^r_h \parallel M^s \subseteq_{I/O} M_1 \parallel \ldots \parallel M_h$, as $M^C \models \psi^C \wedge \neg \delta$ is explicitly checked. Due to the fact that $M_1, \ldots, M_h$ and $M^S$ have disjoint I/O signals, we can conclude that $M^C = M^r_1 \parallel \ldots \parallel M^r_h \parallel M^S \subseteq_{I/O} M_1 \parallel \ldots \parallel M_h$ must also hold.

Therefore, a correct component realization can be checked using common discrete-time model checker to prove $\psi^C \wedge \neg \delta$ for $M^C$ if the restricted I/O refinement for the port refinements hold. To ensure that $M^C$ refines each of the role protocols associated to its ports, we propose to use syntactical refinement rules instead of an explicit verification step (see Section 3.2.3). The required elements and resulting scenario is depicted in Figure 9.

(b) Component Synthesis

Following the in Section 3.2.3 (c) outlined approach for the synthesis of a maximal refinement, a component behavior that refines the pattern roles and fulfills an additional given property can be automatically derived.

For a component $C = (M, \Psi, M^C)$ the behavior $M^C$ can be derived from $M_1 \parallel \ldots \parallel M_h$, the parallel product of the realized pattern roles by looking for the maximal refinement that also fulfill the associated required component invariant $\psi^C$. All required elements and the resulting synthesis scenario are depicted in Figure 10.
Component with $h_j$ ports and behaviour $M^C_j$

Overall behaviour:

$$M^C_j = M^r_{j,1} || \ldots || M^r_{j,h_j} || M^a_j$$

Refinement by construction:

$$\forall 1 \leq m \leq h_j : M^r_{j,m} \subseteq_{I/O} M^C_{j,m}$$

$$M^C_j \models \neg \delta \Rightarrow M^C_j \subseteq_{I/O} M^r_{j,1} || \ldots || M^r_{j,h_j}$$

Figure 9: Elements and view of a port-refined component

Component with $h_j$ ports and behaviour $M^C_j$

Synthesis conditions:

$$M^C_j \subseteq_{I/O} M^r_{j,1} || \ldots || M^r_{j,h_j}$$

$$M^C_j \models \psi^C_j$$

Figure 10: Elements and view of a synthesized component

As the synthesized automaton is only the synchronization skeleton (cf. [EC82]) of the required component realization, besides the synthesis also the more detailed component behavior including its functional behavior and data model remains to be realized.

4.4 Sequence of Design and Verification Steps

Based on the semantic definition in the previous subsection, our approach suggests a particular sequence of integrated design and verification activities organized into the following steps:

(1) design the patterns and their roles,

(2) verify each pattern,

(3) design the components,

(4) verify each component, and

(5) compose the system using the components and patterns.

Note that steps 1 and 2 have to be repeated for every required pattern. When steps 3 and 4 have already been performed with incomplete sets of patterns, the additional roles have to be added to the component automata. Step 5 finally ensures correct semantical composition by a correct
syntactical composition. After each verification we may either proceed with a redesign step or
continue with design or verification activities for another pattern or component.

The possibility to synthesize a maximal refinement w.r.t. a property for a component ex-
plained in Section 4.3.2 results in the following adjusted sequence of steps:

(3’) synthesize the components by refining the parallel roles associated to each port using the
conjunction of their role invariants.

It is to be noted, that the presented additional synthesis step (3’) does additionally ensure the
refinement relation usually proven in step (4). Step (4) thus can be omitted if the synthesis
algorithm has been used.

5 Conclusion and Future Work

The presented approach is based on a restricted notion of patterns only suitable for the considered
domain of hard real-time systems. These patterns further enable to derive the required component
behavior by means of refinement steps for each role, synthesis or manual design and refinement
checking. Finally, the required overall system can be built by only composing components via
our restricted notion of patterns. The approach further permits to verify the system without
building the intractable large state space of the overall system. Instead, each design artifact
(patterns and components) can be verified in ”isolation” first using existing discrete-time model
checking tools. The overall correctness can be derived by only ensuring the syntactically correct
composition of the system.

As the proposed work only builds the formal foundation for the outlined integration within
the current design practice, a number of problems remain to be solved, before the presented
approach can be really employed in practice:

- The specification of real-time behavior with the proposed basic notion of automata is not
  appropriate for direct modeling. Thus, more appropriate notions such as statecharts and
  its time extensions have to be mapped to the formal model rather than directly employing
  the model for modeling purposes. Our proposal to address this problem are real-time
  statecharts [GB03].

- The specification of required system properties with temporal logic is not intuitive and thus
  errors within such specifications are also a major obstacle. Even worse, most designers
  have great difficulties to comprehend the meaning of such formulas. Therefore, temporal
  logic is not an appropriate specification language. Again more appropriate notations for
  the software engineer which are then mapped to temporal logic have to be employed.
  Currently, we use an temporal extensions of OCL named RT-OCL [FM02] that consider
  real-time issues using the system states.

- To complete the cycle of model specification, property specification, and checking of the
  property for the model, the feedback provided by the verification (model checker) has to be
  adjusted to an appropriate abstraction and model level also. Therefore, a counterexample
as well as the not fulfilled part of the property must be represented to the software engineer in such a manner that the connection to the specified model and properties are directly visible. Here we explore whether the generation of shortest paths leading to the errors as well as their animation can improve the usability of the model checking results.

- Last but not least, we have to evaluate the proposed pattern based development and verification approach. In [GFS+03] first results about the application within the collaborative research center 614 of the German National Science Foundation (DFG), titled ”Self-optimizing Concepts and Structures in mechanical Engineering”,

4 which includes 12 research groups from mechanical engineering, electrical engineering, information and computer science and mathematics, is presented. While the general vision of this collaborative research center is to develop concepts and methods to build mechatronics products with inherent intelligence, which react autonomously and flexibly to changing environment and operation conditions, the report uses the more concrete example, a self-optimizing version of the software for the railcab research project5. The particular problem studied, is to reduce the energy consumption due to air resistance by coordinating the autonomously operating shuttles in such a way that they build convoys whenever possible. Such convoys are built on demand and require a small distance between the different shuttles such that a high reduction of energy consumption is achieved. As the coordination between speed control units of the shuttles becomes a safety-critical aspect and results in a number of hard real-time constraints, which have to be addressed when building the control software of the shuttles. As a running example within that report we consider a simplified version of this shuttle coordination problem, namely we assume that only convoys of two shuttles are formed.

We further plan to extend the presented approach such that besides the discrete-time model also a continuous time model is covered. This requires that the whole calculus for compositional reasoning is transferred into the continuous time domain.

Acknowledgements

The author thanks Björn Axenath, Martin Hirsch, Ekkart Kindler, and Daniela Schilling for their comments on earlier versions of the report.

References


4http://www.sfb614.de


REFERENCES


