# Towards the Compositional Verification of Real-Time UML Designs*

Holger Giese, Daniela Schilling†, Matthias Tichy,
Sven Burmester†, and Wilhelm Schäfer
Software Engineering Group
University of Paderborn
Warburger Str. 100
D-33098 Paderborn, Germany
[hg|das|mtt|burmi|wilhelm]@upb.de

Stephan Flake
Heinz Nixdorf Institute
University of Paderborn
Fürstenallee 11
D-33102 Paderborn, Germany
flake@upb.de

July, 2003

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

†supported by the International Graduate School of Dynamic Intelligent Systems.

**Abstract**

Current techniques for the verification of software as e.g. model checking are limited when it comes to the verification of complex distributed embedded real-time systems. Our approach addresses this problem and in particular the state explosion problem for the software controlling mechatronic systems, as we provide a domain specific formal semantic definition for a subset of the UML 2.0 component model and an integrated sequence of design steps. These steps prescribe how to compose complex software systems from domain-specific patterns which model a particular part of the system behavior in a well-defined context. The correctness of these patterns can be verified individually because they have only simple communication behavior and have only a fixed number of participating roles. The composition of these patterns to describe the complete component behavior and the overall system behavior is prescribed by a rigorous syntactic definition which guarantees that the verification of component and system behavior can exploit the results of the verification of individual patterns.

# Contents

# 1 Introduction

Software has become an intrinsic part of increasingly complex distributed embedded real-time systems which are also called (distributed) mechatronic systems. In many cases these systems are used in a safety-critical environment and implement themselves so-called safety critical applications. Consequently, the software controlling these systems has to undergo a rigorous verification and testing process.

Current techniques for the verification of software as e.g. model checking or theorem proving are limited when it comes to the verification of complex systems. The main reason for this scalability problem is the usually enormous state space to be considered when verifying a complete model of the whole system behavior.

We propose a new approach based on current model checking techniques to address the scalability problem of these techniques. This approach suggests to model the software by using the UML 2.0 component model and the corresponding definition of ports and connectors. However, we provide a formal semantic definition of these concepts and an integrated sequence of design steps. These steps prescribe how to compose complex software systems from domain-specific patterns which model a particular part of the system behavior in a well-defined context. Patterns in turn are defined by roles (later becoming component ports) and their corresponding connectors.

The correctness of these patterns can be verified individually based on their well-defined and usually simple communication behavior and the limited number of participating roles such that state explosion is by and large avoided. The composition of these patterns to describe the complete component behavior and the overall system behavior is prescribed by a rigorous syntactic definition. This guarantees that the verification of component and system behavior can exploit the results of the verification of individual patterns. Thus, the verification does not need to consider the complete state space of the system behavior model in one verification step.

The formal semantic definition of the UML 2.0 concepts also takes some domain-specific characteristics into account (like the pattern definition) which again reduces the possible state space of the verification process. The domain-specific characteristics exploited apply to a large class of mechatronics systems and thus the approach is applicable to a wide variety of such systems. In principle, the characteristics assume a very limited number of possible side-effects when performing a system operation which can be guaranteed when a strictly hierarchical system architecture is assumed.

In the next section we review some related work on model checking and design of real-time systems. Then, the specific domain of mechatronic systems and the employed shuttle case study are described in Section 3. The semantics of the employed behavioral real-time model is introduced and formally defined in

Section 4. Afterwards, Section 5 presents the proposed design approach for complex mechatronic systems. Compositional verification of the simplified example is presented in Section 6. In Section 7, we show that the employed local verification steps do indeed ensure that the required constraints also hold for the resulting global system. The paper closes with some final conclusions and an outlook on future work.

# 2 Related Work

A number of different approaches [24, 2, 9, 15] for the object-oriented development of *real-time applications* exists. The most prominent of these approaches that seems to become soon part of the main UML stream is ROOM [24]. Currently, the UML 2.0 proposal of the main tool vendors [21] includes the basic ROOM concepts using the notation of UML/RT [25].

To model complex, physical, possibly distributed architectural objects, components are the appropriate choice in UML 2.0. Components are a specialization of UML encapsulated classifiers, which correspond to UML/RT capsules and the ROOM concept of actors. Components interact with their environment only through signal-based boundary objects called ports. UML ports can have multiple attached UML interfaces denoting the syntactical interface. Each port plays a particular role in a collaboration that the component has within its context. Additionally, UML connectors, which correspond to UML/RT connectors and ROOM bindings, are signal-based communication channels that interconnect multiple ports.

In UML/RT, the notion of a protocol with protocol roles is employed to describe the behavior expected for a set of ports and their connector. It is expected that the connected capsules later respect these protocols. In a sense, a protocol captures the contractual obligations that exist between capsules. The UML 2.0 proposal does not explicitly support this concept, but instead permits to assign a behavior to a UML connector as a ”contract” (see [21, p. 111]).

In contrast to this rather loosely related set of concepts, our approach describes *all* collaborations via a connector and multiple ports in form of a reusable pattern. These patterns are further used to derive the required component behavior in a process that integrates these design activities with verification.

Another thread of development is the *UML Profile for Schedulability, Performance, and Time* [20], which defines time models and additionally allows to attach real-time system specific attributes to classes such as schedulability parameters or quality of service (QoS) characteristics.

However, non-functional system properties w.r.t. dynamic (real-time) behavior are only rudimentary supported in the mentioned real-time approaches for UML.

To overcome this deficiency, temporal extensions of the Object Constraint Language (OCL) have been proposed. OCL has been primarily developed to specify invariants attached to classes and pre- and postconditions of operations. By introducing additional temporal logic operators in OCL (e.g., eventually, always, or never), modelers are able to specify required behavior by means of temporal restrictions among actions and events, e.g., [4]. Temporal extensions of OCL that consider real-time issues have been proposed for events in OCL/RT [6] and for states in RT-OCL [11].

Most of the currently existing work on *model checking* does not consider time-dependent behavior. To tackle such real-time systems, we can make use of model checking techniques that have been extended to verify state transition systems with an explicit notion of time. In this context, we can distinguish two different approaches.

On the one hand, common untimed model checking techniques have been extended to cope with timing aspects. The underlying semantics assume a global clock with discrete time. For timed property specification, extensions of the future-oriented branching-time *Computation Tree Logic* (CTL, see [8]) have been developed, e.g. , RTCTL (real time CTL [10]) or CCTL (clocked CTL [22]). Tools following this approach are VERUS [5] and RAVEN [22].

On the other hand, a more general approach is based on *timed automata* by Alur et al. [1]. In timed automata, time is represented by (an arbitrary number of) real value clocks which can be used to measure time differences w.r.t. a global notion of time. Properties are expressed by Timed CTL (TCTL), an extension of CTL with dense-time semantics. Tools that base upon timed automata are KRONOS[26] and UPPAAL[3]. Note that UPPAAL only supports a limited subset of TCTL dedicated to reachability analysis.

The later employed notion of syntactical refinements permits us to choose any real-time model checker. We have chosen to use RAVEN because we do not need the full power of timed automata, as we assume a minimal time unit for executions of actions and state transitions which leads to a discretization of time. Additionally, we need to be able to verify general properties and cannot restrict on reachability analysis. Under these premises, RAVEN exhibits a better scalability than the other discrete time model checkers, especially when large delay times appear. For an according comparison of RAVEN with VERUS, see [23].

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. [7]). A number of modular and compositional verification approaches have therefore been proposed.

In [17], similar to our approach the decomposition of a system is exploited to permit modular model checking of the system. The notion of decomposition into features in [17] is limited to the sequential case and thus does not address the

classical state explosion problem. It cannot be employed for complex mechatronic systems, because in this domain parallel composition is required.

One particular compositional approach is the assume/guarantee paradigm [19]. Model checking techniques that permit compositional verification following the assume/guarantee paradigm have been developed [8, p. 185ff].

Our approach also follows the assume/guarantee paradigm, but in contrast to current proposals it exploits information available during the design process in form of pattern role protocols to derive the required additional assumed and guaranteed properties automatically rather than manually as in [8]. Moreover, we employ a more restricted notion of refinement which also excludes deadlocks, whereas in [8] only a subset of CTL restricted to the $A$ path quantifier (called ACTL) can be applied. Note that ACTL cannot be used to compositionally exclude deadlocks, because deadlock freedom is $(AG\ (EX\ true))$ in CTL which is not in ACTL.

# 3 Application Domain

Our approach has been developed within the collaborative research center 614 of the German National Science Foundation (DFG), titled "Self-optimizing Concepts and Structures in mechanical Engineering" which includes 12 research groups from mechanical engineering, electrical engineering, information and computer science and mathematics.[1]

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.

As a concrete example, a self-optimizing version of the software for the railcab research project[2] has to be developed which aims at using a passive track system with intelligent shuttles that operate individually and make independent and decentralized operational decisions. Shuttles either transport goods or up to approx. 10 passengers.

The vision of the railcab project is to provide the comfort of individual traffic concerning scheduling and on-demand availability of transportation as well as individually equipped cars on the one hand and the cost and resource effectiveness of public transport on the other hand.

The infrastructure of this shuttle-based transportation system is built by satellite-supported positioning and a wireless communication network to enable communication between shuttles and stationary installations. The modular railway system further combines sophisticated undercarriages with the advantages of

[1] http://www.sfb614.de

[2] http://www-nbp.upb.de/en/index.html

new actuation techniques as employed in the Transrapid[3] to increase passenger comfort while still enabling high speed transportation and (re-)using the existing railway tracks.

One particular problem 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.

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 this paper we consider a simplified version of this shuttle coordination problem, namely we assume that only convoys of two shuttles are formed.

One main requirement of the shuttle controller software in this example is to ensure that no rear-end collision happens when the first shuttle has to brake suddenly e.g. in case of an emergency. The second shuttle should however still keep a minimal distance to the one ahead of it for the reasons mentioned above. This distance is computed based on the delay, the speed, the weight, the maximum force of the brakes etc. and is a compromise between safety and cost-effectiveness concerning energy consumption.

Controlling the distance cannot be done locally by a shuttle alone using distance sensors because (1) the distance cannot be always measured directly (e.g. in a turn) and thus the wireless communication network has to be used to propagate position and speed, and (2) reducing the distance in convoy mode means to e.g. reduce the speed or the brake force in a coordinated fashion between the two shuttles.

So in any case the controller software of a shuttle has to communicate with the other one in order to decide what to do when forming a convoy. Then they can decide on a common strategy like reducing velocity, increasing the distance, decreasing brake force or a combination of all of them. One possible simple solution which is used in this paper, is that the first shuttle will always brake with limited force when it is in convoy mode.

This example also illustrates that real-time model checking is hopeless in the general case, because it means to ensure the correct cooperation between a usually large number of shuttles. This applies of course in general to complex arbitrarily structured distributed software systems.

However, by exploiting some domain specific restrictions which are common to most of these systems our approach becomes feasible. These restrictions are (1)

[3] http://www.transrapid.de/en/index.html

the usual clock synchronization assumption which is common to many systems and means that time is progressing equally fast in any system component, (2) a discrete time model suffices to model all time depending constraints, because the underlying infrastructure (hardware and possibly a real-time operating systems) does not react infinitly fast, (3) a layered system architecture which guarantees that an autonomous unit like a shuttle reacts in a local environment and the interfaces to its environment are strictly defined (as e.g. a shuttle trying to form a convoy has to interact only with one other shuttle and not maybe with a third one which is a few kilometers away).

Restrictions 1 and 2 allow us to assume a rather simple notation for time (see next section). Restriction 3 is the reason that usually only relatively simple patterns have to be constructed, i.e. patterns with simple coordination protocols between roles, limited numbers of input signals and a fixed number of roles.

# 4 Real-Time Semantics

We have to formally define the semantics of the employed UML concepts such that the restrictions and requirements of our application domain observed in the last section are fulfilled. The request for verifiability further requires that the employed concepts have a rigorous foundation. Therefore, we also present formal definitions for the employed notion of automata, parallel composition, and refinement.

## 4.1 Statecharts Semantics

We employ time-annotated statecharts to describe required real-time behavior. For the considered domain of mechatronic systems, the rather complex micro step semantics of UML statecharts is not necessary. Instead, in each state machine cycle only a single transition is fired. Such a semantics has already successfully been employed in a similar domain [16] for the untimed case. Note that due to our simplified statechart semantics most of the problems w.r.t. compositionallity described in the literature can be avoided (cf. [18]).

For our approach, more emphasis is put on timing aspects to ensure that synthesis of hard real-time code is indeed possible [12]. Therefore, we make the additional assumption that each transition in a statechart has a worst-case execution time that is lower than the assumed minimal time amount (1 msec in our example). Thus, we assume that every fired untimed transition can be executed within one time unit or otherwise has to be manually split into a sequence of transitions. Instead using a sequence of untimed transitions, an after() statement might be used.

We use the scheme x.signal to denote that a signal from/to a specific role or port x is received/sent. Whether it is sent or received is denoted by its appearance as trigger (before the slash) or action (behind the slash).

To specify that a statechart will only remain in a specific state for at most $x$ time steps, we use the syntactical shortcut atMost: $x$ msec. It is equivalent to an after(x sec) transition with a special deadlocking target state and emulates time invariants for states as employed in timed automata (see [1, 12]).

Modelling the behavior of a pattern role requires that all alternatives of a later realization can be specified. Thus non-determinism due to multiple alternatively enabled transitions for a single state naturally follows. A second form of non-determinism occurs when several alternatives for the delay of a transition are required. Intervals instead of a concrete single delay value as parameter of an after() statement are used to denote that the transition may take any of the specified time units before firing.

The formal semantics of the described time-annotated statecharts is defined by mapping them to a finite state transition system in form of extended Kripke structures (called I/O-interval structures [22]) as employed by the RAVEN model checker. We present here only a rather simplified version of this finite state transition model where discrete time is mapped to single states and transitions. This automata model is sufficient to permit the understanding of the underlying behavior model and to proof that the compositional verification is correct. It is to be noted, that the real-time model checker will of course use a more compact representation of time to reduce the number of explicit considered states.

**Definition 1.** *An* automaton *is a* 5*-tuple* $M = (S, I, O, T, Q)$ *with a finite set* $S$ *of states, input signals* $I$*, output signals* $O$*, a set of transitions* $T \subseteq S \times \wp(I) \times \wp(O) \times S$*, and the initial state set* $Q$*. A* run *is a sequence of states* $s_1, s_2, \ldots,$ *where for each* $i \geq 1$ *exists* $(s_i, A, B, s_{i+1}) \in T$*. We further require that for each* $s \in S$ *there is at least one finite run* $s_1, s_2, \ldots s_n$ *with* $s_1 \in Q$ *and* $s = s_n$*.*

The time semantics of an automaton is simply that each transition takes exactly one time unit.

For convenience we use in the following $S_i$, $I_i$, $O_i$, $T_i$, and $Q_i$ to denote the corresponding elements of $M_i$. Two automata $M$ and $M'$ with distinct input and output sets ($I \cap I' = \emptyset$ and $O \cap O' = \emptyset$) are further called *composable*. If also $I \cap O' = \emptyset$ and $O \cap I' = \emptyset$ holds, they are even *orthogonal* to each other. An automaton $M$ with $I = O$ is further called *closed*.

## 4.2 Property Specification

Properties which should hold for a specific model and which have to be checked by the model checker are specified by using the state-oriented real-time OCL variant

RT-OCL [11]. RT-OCL is formally defined by a mapping to the temporal logic CCTL used in the RAVEN real-time model checker.

To reflect the dependencies which result from local or shared name spaces within the OCL and RT-OCL constraints, the CCTL equivalents of RT-OCL constraints ($\phi$) and OCL invariants ($\psi$) will use a shared set of atomic propositions $P$. An automaton $M_i$ and any of its states $s \in S_i$ is annotated with all propositions in $P_i \subseteq P$ which they fulfill using a labelling function $L_i : S \rightarrow \wp(P_i)$. Thus an automaton $M_i = (S_i, I_i, O_i, T_i, Q_i)$ is accordingly extended to a 6-tuple $M_i = (S_i, I_i, O_i, T_i, L_i, Q_i)$. The label set $\mathcal{L}(M_i)$ denotes the set of all by the labelling considered propositions $P_i$. $\mathcal{L}(\phi)$ and $\mathcal{L}(\psi)$ denote the subsets of the basic proposition set $P$ that is employed within the formulas.

Finally, for sake of simplification of the following formal definitions, we omit any syntactical details of CTL and CCTL and write $M \models \phi$ when an automaton $M$ fulfills a constraint or invariant $\phi$. The special symbol $\delta$ is used to denote that a *deadlock* (a state without any outgoing transition) can be reached. $M \models \neg\delta$ thus denotes that $M$ does not contain any deadlocks.

## 4.3 Parallel Composition

In our application domain the composition of multiple components requires their parallel execution. As we model time explicitly and in a discrete manner, the required notion of parallel composition must result in the *synchronous execution* [8] of all systems running in parallel.

The communication is formalized by *synchronous communication* such that sending and receiving happens within the same time step. Consequently, the asynchronous event semantics of statecharts is modelled by explicitly defined event queues (channels) given in form of additional automata. These explicit models of the event queues are required anyway to take the QoS characteristics of each connection into account.

**Definition 2.** *For two automata* $M = (S, I, O, T, L, Q)$ *and* $M' = (S', I', O', T', L', Q')$ *which are composable to each other (*$I \cap I' = \emptyset$ *and* $O \cap O' = \emptyset$*), we define their* parallel composition *denoted by* $M \| M'$ *as the automaton* $(S'', I'', O'', T'', L'', Q'')$ *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* $A'' = A \cup A'$ *and* $B'' = B \cup B'$*. Additionally,* $(A \cap O') = (B' \cap I)$ *and* $(A' \cap O) = (B \cap I')$ *must hold.* $S''$ *and* $T''$ *are further adjusted to exclude all non reachable state combinations and transitions. The labelling* $L''$ *for* $(s, s') \in S''$ *is easily 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 non local input and output signals are simply the union of both automata.

## 4.4 Automata Refinement

Our restricted notion of components means that they are derived by refining the role protocols from all the patterns they are participating in. A component is thus built by parallel composition of port statecharts and an additional synchronization statechart for further internal coordination. Thus, we require an appropriate notion for refinement which is essentially a restricted version of simulation which additionally preserves reactivity.

**Definition 3.** *An automaton $M = (S, I, O, T, L, Q)$ is a* refinement *of automaton $M' = (S', I', O', T', L', Q')$ ($M \sqsubseteq M'$) iff a relation $\Omega$ exists with $\Omega \subseteq S \times S'$ and $\forall q \in Q\ \exists q' \in Q' : (q, q') \in \Omega$ and for all $(s_1, s_1') \in \Omega$ must hold:*

$$\forall (s_1, A, B, s_2) \in T \quad \exists (s_1', A, B, s_2') \in T' : (s_2, s_2') \in \Omega, \tag{1}$$

$$\forall (s_1', A', B', s_3') \in T' \quad \exists (s_1, A', B', s_3) \in T. \tag{2}$$

*For given labelling functions $L$ and $L'$ we require that they are also 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 1 further ensures that a related transition in $M'$ exists that again leads to an appropriate state pair in $\Omega$. Therefore, $\sqsubseteq$ implies simulation ($\preceq$). Equation 2 then further ensures that for each in a state offered pair of $I/O$ signal sets in $M'$ a corresponding transition offering the same pair of $I/O$ signal set is provided in its refinement $M$. However, the condition does not itself require that $s_3$ and $s_3'$ build a pair contained in $\Omega$.

To also build a refinement notion that permits the refined behavior to extend the original one, a restriction operator $|$ is required to abstract from additional signals. For an automaton $M = (S, I, O, T, L, Q)$ we define its $I/O$ *and labelling restriction* for $I''/O''/\mathcal{L}''$ denoted by $M|_{I''/O''/\mathcal{L}''}$ as the automaton $(S', I', O', T', L', Q')$ with $S' = S$, $I' = I \cap I''$, $O' = O \cap O''$, $L'(x) = L(x) \cap \mathcal{L}''$, $Q' = Q$, and $(s_1', A', B', s_2') \in T'$ iff $(s_1, A, B, s_2) \in T$ exists with $A' = A - I''$ and $B' = B - O''$.

When for two basic automata $M = (S, I, O, T, L, Q)$ and $M' = (S', I', O', T', L', Q')$ hold that $M|_{I'/O'/\mathcal{L}(M')}$ is a refinement of $M'$ we further name $M$ to be a *restricted refinement* of $M'$ ($M \sqsubseteq_{I/O} M'$). This restricted refinement adjusts the considered signals and can be further used to characterize if an automaton is a correct concretization of another one.

# 5 Design Steps

Based on the semantic definition in the previous section, 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 refining the roles associated to each port, (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, additional parallel statecharts that refine the additional roles have to be added incrementally. Step 5 finally ensures correct semantical composition by a correct syntactical composition.

In the remainder of this section, modelling steps 1, 3, and 5 are described, whereas verification steps 2 and 4 are presented in Section 6.

## 5.1 Pattern Definition

In our approach a pattern comprises of a set of roles that interact only via ports and a related connector that connects those ports. We further have the restriction that for each pattern we have to specify a protocol automata and OCL invariants for each role. An overall constraint in form of a RT-OCL formula is also possible. While usually 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 4.** *A* pattern $P$ *is a* $4$*-tuple* $(\mathcal{M}, \Psi, \phi, M^P)$ *with a set* $\mathcal{M}$ *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* $M^P$*.*

For multiple patterns $P_1, \ldots, P_n$ we refer to their constraints as $\phi_i^P$ and connector automata as $M_i^P$.

In our example we consider a pattern for the convoy behavior of two participants. This DistanceCoordination pattern is intended to realize safe coordination of the shuttle distance. It consists of two roles FrontRole and RearRole denoting the relative position within a convoy. The roles are interconnected via a connector representing the wireless communication network and its QoS characteristics. We will further have Shuttle components as visualized in Figure 1 that realize the pattern to ensure a safe coordination of the required minimal distances.

Figures 2 and 3 show the main parts of the statecharts for the pattern role protocols. The two roles model the discrete behavior of two participants (which

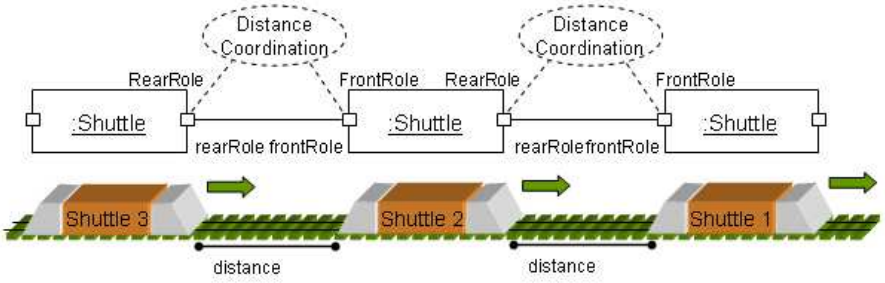


Figure 1: Patterns ensure appropriate shuttle distances

are in our example the shuttles), where they describe the part of the rear resp. the front position in a potential convoy.

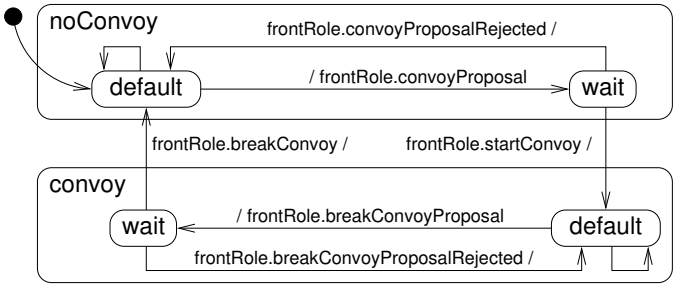


Figure 2: Statechart of the RearRole::Main role

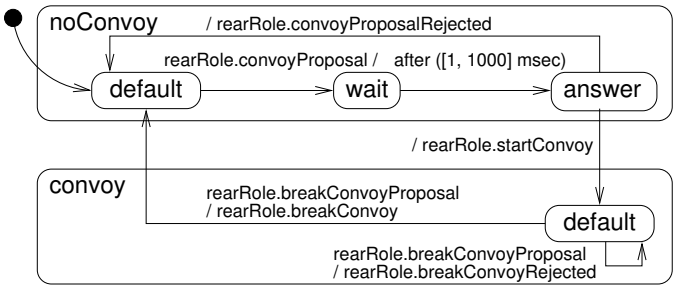


Figure 3: Statechart of the FrontRole::Main role

Initially both roles are in state noConvoy::default, which means that they are not in a convoy. The rear role non-deterministically chooses whether to propose forming a convoy or not. After choosing to propose a convoy, a message is sent to the other shuttle resp. its front role. The front role chooses non-deterministically to reject or to accept the proposal after max. 1000 msec. In the first case, both statecharts revert to the noConvoy::default state. In the second case, both roles switch to the convoy::default state.

Eventually the rear shuttle non-deterministically chooses to propose a break of the convoy and sends this proposal to the front shuttle. The front shuttle chooses non-deterministically to reject or accept that proposal. In the first case, both shuttles remain in convoy-mode. In the second case, the front shuttle replies by an approval message, and both roles switch into their respective noConvoy::default state.

Additionally, the front shuttle periodically sends position and speed data to the rear shuttle as modeled in the statecharts FrontRole::Ping and RearRole::Pong shown in Figures 4 and 5.

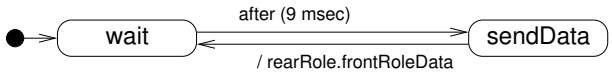


Figure 4: FrontRole::Ping statechart

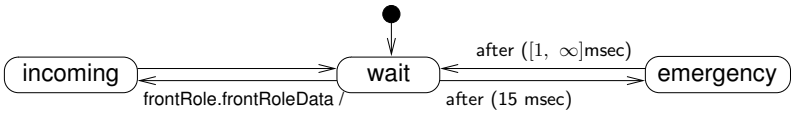


Figure 5: RearRole::Pong statechart

This protocol ensures that the rear shuttle receives the required data of the front shuttle to adjust its distance. After a timeout of 15 msec without receiving data, the rear shuttle assumes that either the front shuttle or the connector has failed and thus switches into the emergency state to indicate the problem.

The statechart resulting from the parallel composition of FrontRole::Main and FrontRole::Ping then describes the protocol $M_1$ for the FrontRole role. The statechart $M_2$ for the RearRole role is accordingly built by the parallel composition of statecharts RearRole::Main and RearRole::Pong.

For the connector which represents the wireless network we do not apply an explicit statechart, but instead specify its QoS characteristics such as throughput, maximal delay etc. in the form of connector attributes. The required automaton $M^P$ is automatically derived from these attributes. In our case study, we assume that the connector forwards incoming signals with a delay of 1 up to 5 msec. The connector is unsafe in the sense that it might fail at any time, such that we set our specific QoS characteristic reliable to false.

To provide fail safe behavior, the following RT-OCL constraint named $\phi$ must hold (cf. Section 3). It demands that (a) a combination of role states where the front role is in state noConvoy and the rear role in state convoy is not possible, and (b) data that is sent must be received within 6 msec (i.e., 5 msec + 1 msec to get to state incoming) or the connector has failed.

```
context DistanceCoordination inv:
    not (self.oclInState(RearRole::Main::convoy) and
         self.oclInState(FrontRole::Main::noConvoy))
  and
   (self.oclInState(FrontRole::Ping::sendData)
    implies
      self@post(1,6)->forAll(p:OclPath |
        p->exists(c:OclConfiguration |
                  c->includes(RearRole::Pong::incoming)
                  or c->includes(Channel::fail))))
```

We ensure by construction that the emergency state of RearRole::Pong is entered if the connector has failed because of the timout transition labeled after(15 msec) from state wait to emergency in Figure 5.

For an abstract property CanBrakeFully which each realizing component has to provide, we additionally require for any implementation of the rear role that being in state convoy implies that CanBrakeFully holds. In contrast, for any implementation of the front role, state convoy requires that CanBrakeFully does not hold. The following OCL role invariants $\psi_1$ and $\psi_2$ are used to describe these restrictions which apply to any component which realizes the specific role. We here abstract from the actual realizing components and ports by means of a (non-standard) schema definition that is syntactically denoted by enclosing angle brackets.

```
context <component> inv:
  <frontRole>.oclInState(convoy) implies
    not self.CanBrakeFully

context <component> inv:
  <rearRole>.oclInState(convoy) implies
    self.CanBrakeFully
```

The pattern thus consists of the two roles FrontRole and RearRole denoted by $M_1$ resp. $M_2$, the role invariants $\psi_1$ and $\psi_2$ for FrontRole resp. RearRole, one RT-OCL constraint $\phi$, and a connector modelling a wireless and thus non reliable network $M^P$. Thus together we have the 4-tuple $P = (\{M_1, M_2\}, \{\psi_1, \psi_2\}, \phi, M^P)$ that formally represents the DistanceCoordination pattern $P$.

Note for further reading that the role specification of the presented example pattern contains an error which will be later used to exemplify our verification approach.

## 5.2 Component Definition

Components are designed by coordinating and refining each role automaton based on Definition 3. The 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. An additional internal

statechart for coordination is used to describe the required coordination. Formally, we can thus define a component as follows:

**Definition 5.** *A* component $C$ *is a triple* $(\mathcal{M}, \Psi, M^s)$ *with a set* $\mathcal{M}$ *of automata* $M_1^r, \ldots, M_h^r$ *refining the realized pattern roles, the set* $\Psi$ *of all associated role invariants* $\psi_1, \ldots, \psi_h$*, and the component internal synchronization automaton* $M^s$*.*

$M^C = M^s \| M_1^r \| \ldots \| M_h^r$ is the overall component automaton, and the component role invariant $\psi^C$ is derived by simply combining the related role invariants ($\psi_1 \wedge \cdots \wedge \psi_h$). For multiple components $C_1, \ldots, C_m$ we refer to their overall component behavior as $M_j^C$ and invariant as $\psi_j^C$.

In our example, the shuttle component must conform to the DistanceCoordination pattern and has to operate as both a RearRole and as a FrontRole. The non-deterministic choice of proposing and accepting the forming of a convoy is specified by the additional statechart $M^s$ for the synchronization of the role refinements (Figure 6). The port statecharts which refine the pattern roles are shown in Figures 7 and 8. Note that both roles FrontRole and RearRole refer to distinct instantiations of the coordination pattern at run-time.

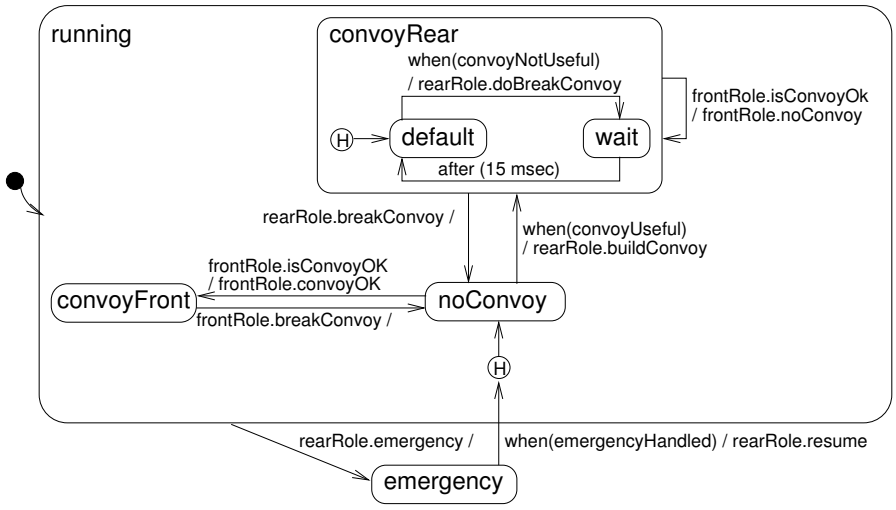


Figure 6: Shuttle synchronization statechart

The shuttle synchronization statechart initiates the building and breaking of a convoy by sending buildConvoy resp. doBreakConvoy to the refined rear role when the guards indicate that it is useful resp. not useful to run in convoy mode. The protocol between the (refined) frontRole and rearRole causes that the front shuttle receives isConvoyOK leading the statechart to switch to convoyFront where it remains until it receives breakConvoy. In case of an emergency (see Figure 9),

the statechart switches to the emergency state. To keep the example simple, the complex procedure required to recover from an emergency state is omitted and we assume a simple guard emergencyHandled to control possible recovery.

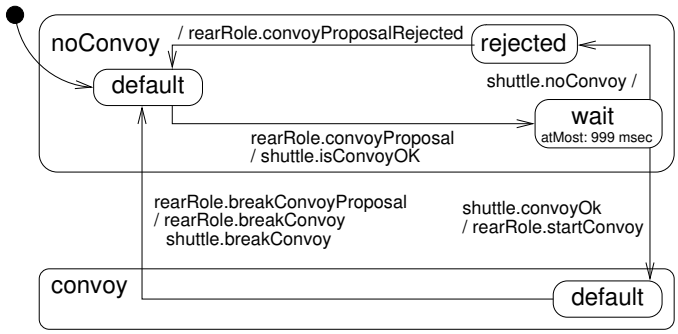


Figure 7: Refined FrontRole::Main statechart

It is important that the refined FrontRole::Main is a correct refinement of the pattern role FrontRole::Main. We have a timing behavior which is either covered by the protocol statechart or will lead to a deadlock when the assumed message from the shuttle synchronization statechart is not received due to the atMost: 999 msec restriction. As we check for deadlocks locally within each component, the second case is not relevant, and in the first case it is sufficient to ensure the required notion of refinement (see Section 7.3). For the refined RearRole::Main statechart, reasoning is analogical.

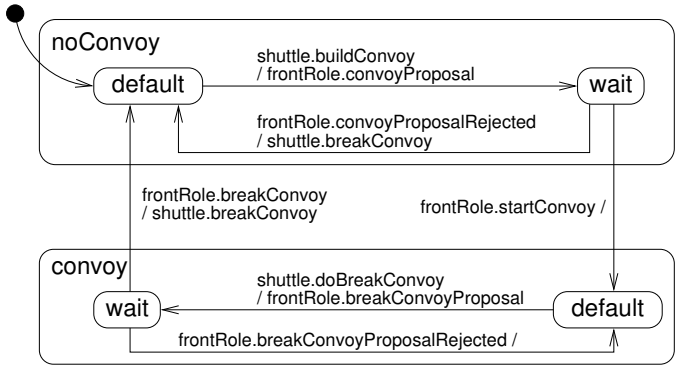


Figure 8: Refined RearRole::Main statechart

Note the emergency state (see Figure 6), which is triggered by emergency signals. This signal is sent by a refinement of the RearRole::Pong statechart. Sending this signal is added as an additional side-effect to the transition which fires if a timeout occurs (see Figure 9).

The port statecharts for FrontRole and RearRole are therefore built by combining Shuttle::FrontRole::Main and Shuttle::FrontRole::Ping to build $M_1^r$ resp. Shut-

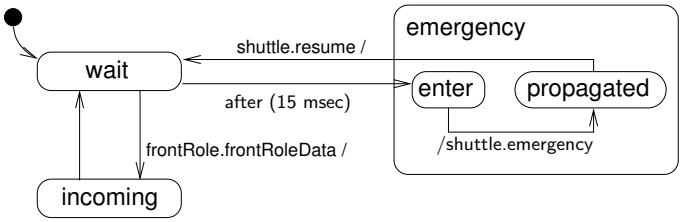


Figure 9: Refined RearRole::Pong port statechart

tle::RearRole::Main and Shuttle::RearRole::Pong to build $M_2^r$. Using the role invariants from the DistanceCoordination pattern, we can build the required triple $C = (\{M_1^r, M_2^r\}, \{\psi_1, \psi_2\}, M^s)$ for this component.

## 5.3 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 (see Figure 10). This can be formally defined as follows:

**Definition 6.** *A* system $\mathcal{S}$ *is a triple* $(\mathcal{P}, \mathcal{C}, map)$ *with a set* $\mathcal{P}$ *of patterns* $P_1, \ldots P_n$*, a set* $\mathcal{C}$ *of components* $C_1, \ldots, C_m$*, and a bijective mapping* $map$ *which assigns to each component port the related unique pattern role. The* syntactical correctness *of such a system requires that all related automata* $M_1^P, \ldots, M_n^P, M_1^C, \ldots, M_m^C$ *are connected accordingly by* $map$ *such that all roles are realized by the component ports.*

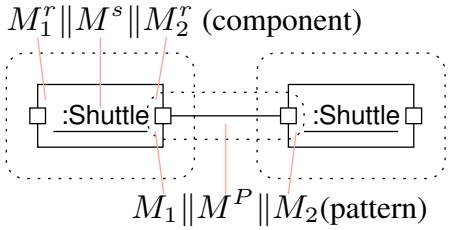


Figure 10: Structural model with related automata

In our example we can thus build arbitrary complex combinations of shuttle components connected via the DistanceCoordination pattern using multiple instances which are accordingly adjusted to permit their composition (renaming of signals etc.). Therefore, instead of $n$ patterns and $m$ components, we will usually only have $n'$ and $m'$ different patterns resp. components within a single system.

# 6 Compositional Verification

In this section, we describe in more detail the design steps 2 and 4 as outlined in the beginning of Section 5.

## 6.1 Pattern Verification

In design step 2, we verify whether the behavioral requirement specified by means of RT-OCL hold for a pattern. If the requirement holds, the pattern is named *correct*. Formally, a pattern $P = (\mathcal{M}, \Psi, \phi, M^P)$ with a set $\mathcal{M}$ of automata $M_1$, ..., $M_k$ is a *correct pattern* iff:

$$M_1 \| \ldots \| M_k \| M^P \models \phi \wedge \neg\delta \tag{3}$$

This can be verified using a real-time model checker which first builds the model $M_1 \| \ldots \| M_k \| M^P$ and then checks whether the constraint $\phi \wedge \neg\delta$ 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. The mentioned domain restrictions 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. Thus, the required verification becomes possible when the state space of each single pattern is not too large.

For our example we generate synchronous automata from the statecharts for FrontRole::Main, FrontRole::Ping, RearRole::Main, RearRole::Pong. Additionally, an automaton for the implicit connector with QoS characteristics delay = [1,5] msec and reliable = false is built that forwards incoming signals to their destination. Then we check whether $\phi \wedge \neg\delta$ holds.

It turns out that the pattern constraint does not hold because of the following possible execution path (as shown in Figure 11): Assume that both roles are in state convoy and rearRole::Main sent breakConvoyProposal to frontRole::Main. If the front role accepts the proposal it sends the acknowledgement breakConvoy to the rear role and changes to noConvoy superstate. In the case that the connector fails to forward the signal breakConvoy to rearRole::Main, the resulting situation is that frontRole::Main is in state noConvoy, while rearRole::Main is in state convoy. This violates the specified pattern constraint $\phi$.

Thus we have to change the model as e.g. illustrated in Figures 12 and 13. Now, FrontRole::Main initiates the breaking of the convoy by sending the breakConvoyProposal message to RearRole::Main, and breakConvoy is sent from RearRole::Main to FrontRole::Main. Therefore a loss of the reply would lead to the acceptable situation that FrontRole::Main is in convoy mode while RearRole::Main is

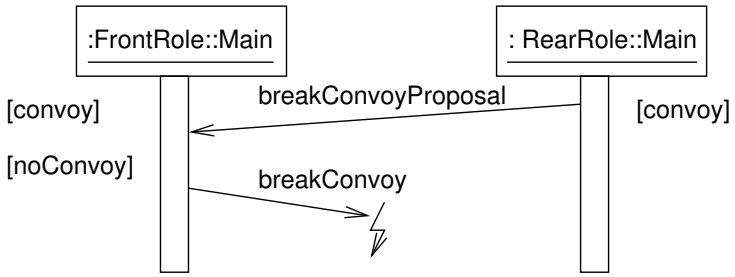


Figure 11: Counter example

in state noConvoy. As this error is detected before the component is actually specified, an adjustment of the component behavior and the synchronization statechart would usually not be necessary.

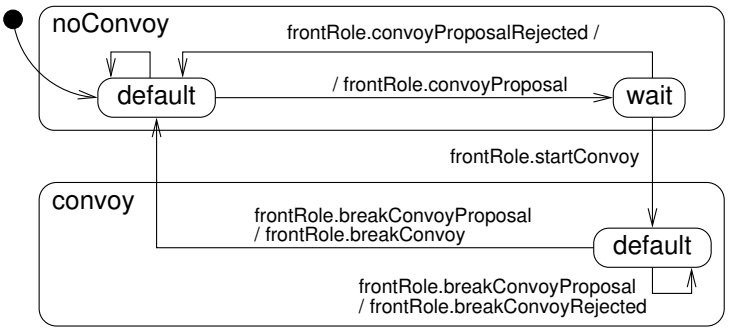


Figure 12: Corrected RearRole::Main statechart

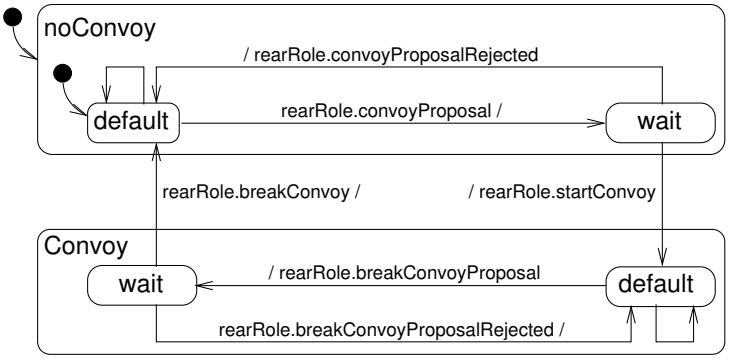


Figure 13: Statechart of the corrected FrontRole::Main role

After verifying the pattern, described by the models from Figures 12 and 13, the components and the synchronization statechart can be specified. Here, the shuttle synchronization sends the doBreakConvoy message to FrontRole::Main in-

stead of RearRole::Main when driving in a convoy is not useful any more (cf. Figure 14). The component behavior is refined in Figures 15 and 16.

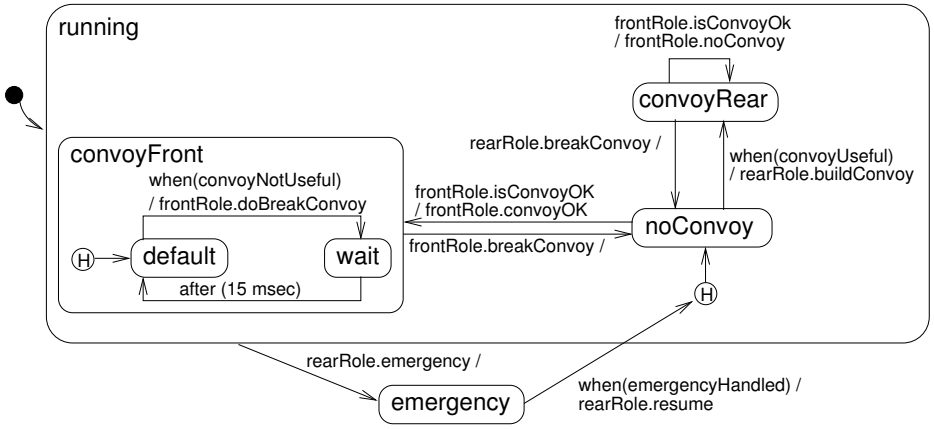


Figure 14: Corrected shuttle synchronization statechart

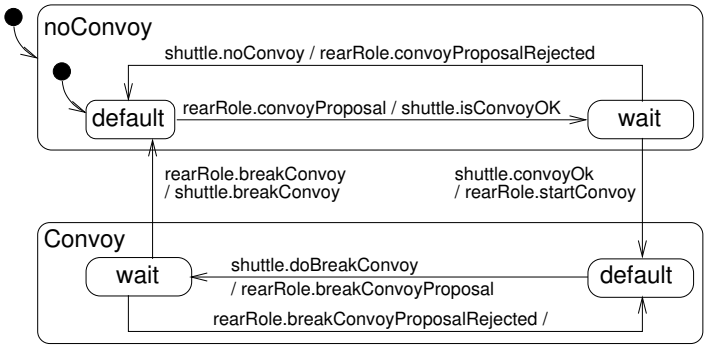


Figure 15: Statechart of the corrected refined FrontRole::Main role

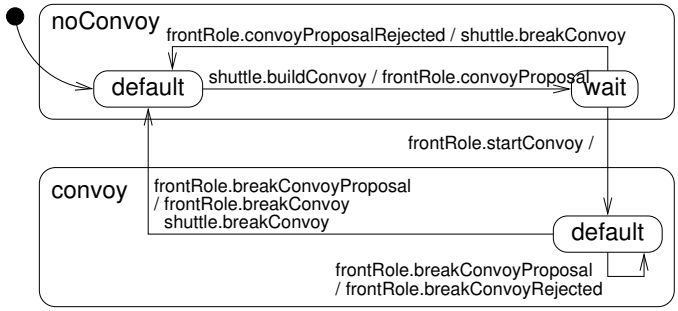


Figure 16: Statechart of the corrected refined RearRole::Main role

## 6.2 Component Verification

Besides the patterns also the components have to be verified. In step 4 of the outlined sequence of design and verification activities, 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. A component $C = (\mathcal{M}, \Psi, M^s)$ with a set $\mathcal{M}$ of automata $M_1^r, \ldots, M_h^r$ is a *correct component* if for each of its refined role behaviors $M_j^r$ and the corresponding original role behavior $map(M_j^r)$ holds:

$$M^C \sqsubseteq_{I/O} map(M_1^r) \| \ldots \| map(M_h^r) \quad \text{and} \quad M^C \models \psi^C \wedge \neg\delta \tag{4}$$

We again can use a real-time model checker to prove $\psi^C \wedge \neg\delta$ for $M^C$. 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 7.3).

Proving the correctness of all $m$ components ($m'$ different ones) 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 $h$ exists in our domain.

The invariant for the shuttle property CanBrakeFully is automatically derived from the role invariants $\psi_1$ and $\psi_2$ as given in Section 5.1. In the resulting invariant, frontRole and rearRole are now concrete names for navigation to the associated ports according to $map$. Generally, the names can be different from the schemas specified for each role before.

```
context Shuttle inv:
frontRole.oclInState(convoy) implies
    not self.CanBrakeFully
  and
    rearRole.oclInState(convoy) implies
      self.CanBrakeFully
```

The synchronization statechart fulfills the abstract CanBrakeFully property for all states except the convoyFront state. Therefore the two implications above are not contradictionary for shuttle components, as due to our design goal never both roles are in convoy mode at the same time.

## 6.3 System Verification

Due to the compositional nature of our approach, an additional 6th step to perform verification for the overall system after its composition in step 5 is not required. In the remainder of this section, we define our notion of a semantically correct system and informally argue why it can be achieved via syntactical correctness only.

For $\mathcal{S} = (\mathcal{P}, \mathcal{C}, map)$ with a set $\mathcal{P}$ of patterns $P_1, \ldots P_n$, a set $\mathcal{C}$ of components $C_1, \ldots, C_m$, and a bijective mapping $map$ to be a *correct system*, *semantical correctness* holds iff the pattern constraints $\phi_i^P$ and component invariants $\psi_j^C$ also hold for the system itself:

$$M_1^P \| \ldots \| M_n^P \| M_1^C \| \ldots \| M_m^C \models \phi_1^P \wedge \cdots \wedge \phi_n^P \wedge \neg\delta \text{ and} \quad (5)$$

$$M_1^P \| \ldots \| M_n^P \| M_1^C \| \ldots \| M_m^C \models \psi_1^C \wedge \cdots \wedge \psi_m^C. \quad (6)$$

In Figure 10 we depicted the different models built for verification. 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 nondeterministic context for the components as well as guaranteed behavior of each pattern role.

At the border of these subsets there are always well-defined protocols both sides agreed upon and respect. Due to the specific characteristics of the considered domain the real-time character of these 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 [13]).

For the restricted class of compositional properties (see Definition 7) we can also use the border built by the ports resp. roles to also proof the constraints $\phi_i^P$ and invariants $\psi_j^C$ compositionally.

> A system with only correct patterns and only correct components is semantically correct if all elements are syntactically correct connected (see Theorem 1 in Section 7.2).

The advantage of the compositional approach is that Theorem 1 permits us to verify condition 5 and 6 without building the state space for $M_1^P \| \ldots \| M_n^P \| M_1^C \| \ldots \| M_m^C$. 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 pattern in the overall system can result in one check in $O(\exp(n+m))$ due to the possibly exponential growing product 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 required efforts do only grow linear with the number of different patterns and components which make the approach scalable.

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.

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 which involve more than one component is required, we have to describe the parallel composition of all involved elements within a single pattern or component using either the connector or internal synchronization automata. Then, the for this pattern resp. component proven result will also hold for the overall system using Theorem 1.

# 7 Formalization

This section formally underpins the employed composition verification approach. The approach yields a verification result for the overall system without building its complete state space (for more details see [14]).

## 7.1 Compositional Constraints

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

**Definition 7.** *A constraint $\phi$ is* compositional *iff for any automata $M_1$, $M_1'$, and $M_2$ with $\mathcal{L}(M_2) \cap \mathcal{L}(\phi) = \emptyset$ holds*

$$(M_1 \models \phi) \Rightarrow ((M_1 \| M_2 \models \phi) \vee (M_1 \| M_2 \models \delta)) \text{ and} \tag{7}$$

$$((M_1 \sqsubseteq M_1') \wedge (M_1' \models \phi)) \Rightarrow (M_1 \models \phi) \tag{8}$$

CTL formulas are preserved by the bisimulation equivalence relation, while ACTL formulas are preserved by the simulation preorder ($\preceq$) [8]. The presented refinement implies simulation and thus preserves ACTL formulas also, but in contrast it additionally preserves deadlock freedom:

**Lemma 1.** *For automata $M$ and $M'$ with $M \sqsubseteq M'$ holds $M' \models \neg\delta \Rightarrow M \models \neg\delta$.*

*Proof.* (sketch) Condition 1 ensures that for any $s \in S$ at least one related $s' \in S'$ exists with $(s, s') \in \Omega$. From $M'$ deadlock free follows that $s'$ will have at least one outgoing transition and due to condition 2 $s$ also. Therefore, $M$ is also deadlock free. □

Invariants, upper and lower time-bounds, and ACTL 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 are compositional. That deadlock freedom is also compositional follows by construction for condition 7 and Lemma 1 for condition 8.

Compositionallity can thus been established for the properties required so far during our studies such as deadlock freedom, upper bounds for the maximal delays of message transports, lower bounds for the minimal delays of message transports, and invariants. For example, the according CCTL formula with only $A$ path quantifiers for a maximal delay is for $d$ the maximal delay, $p_1$ the trigger condition, and $p_2$ the required condition: $AG(\neg p_1 \vee (AF_{[1,d]}\, p_2))$. In contrast, temporal logic formulas that demand explicitly that a specific state is eventually reached (abstracting from possible effects of non-determinism) are not preserved.

## 7.2 Compositional Verification Theorem

We also require the property, that composition preserves refinement for the parallel composition.

**Lemma 2.** *For any automaton $M_1$ and an automaton $M_2$ refining automaton $M_2'$ ($M_2 \sqsubseteq M_2'$) 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 form the relation $\Omega$ implied by the refinement $M_2 \sqsubseteq M_2'$ and derive a relation $\Omega'$ required for the refinement $M_1 \| M_2 \sqsubseteq M_1 \| M_2'$ as follows: For all $(s_1, s_2') \in S_1 \times S_2'$ and $(s_2, s_2') \in \Omega$ add $(s_1, s_2)$ to $\Omega'$. Due to the composition of $T$ resp. $T'$ from $T_1$ and $T_2$ resp. $T_2'$ we can easily prove condition 1 and 2. □

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

**Lemma 3.** *For automata $M_1$, $M_2$, and $M_2'$ with $M_2 \sqsubseteq_{I/O} M_2'$, $I_1 \cap (O_2 - O_2') = \emptyset$, $O_1 \cap (I_2 - I_2') = \emptyset$, and $\mathcal{L}(M_1) \cap (\mathcal{L}(M_2) - \mathcal{L}(M_2')) = \emptyset$ and any compositional constraint $\phi$ holds*

$$(M_1 \| M_2' \models \phi \wedge \neg\delta) \Rightarrow (M_1 \| M_2 \models \phi \wedge \neg\delta) \tag{9}$$

*Proof.* Due to $\phi$ and $\neg\delta$ being compositional and Definition 7 we can for $M_2'' = M_2|_{I_2'/O_2'/\mathcal{L}(M_2')}$ conclude that $M_1 \| M_2'' \models \phi \wedge \neg\delta$ or $M_1 \| M_2'' \models \delta$. Due to Lemma 1 and 2 we even have $M_1 \| M_2'' \models \phi \wedge \neg\delta$. From $I_1 \cap (O_2 - O_2') = \emptyset$ and

$O_1 \cap (I_2 - I_2') = \emptyset$ follows that $M_2$ adds to $M_2''$ only I/O that does not interfere with $M_1$ and thus $M_1 \| M_2$ has the same reachable state set and transitions and thus $M_1 \| M_2 \models \neg\delta$. As $\phi$ is only interpreted over states and the labelling is identical for $\mathcal{L}(\phi) \subseteq \mathcal{L}(M_2')$, $\phi$ must also hold and thus condition 9 is proven. □

We have to restrict patterns 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 any $i \neq j$. For different components we also require the label sets to be disjoint ($\mathcal{L}(M_i^C) \cap \mathcal{L}(M_j^C) = \emptyset$). By choosing appropriate labelling functions for the synchronization statecharts 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 (see Section 6.2).

Using the above definitions and proven facts we can derive the following main compositionallity result of our approach.

**Theorem 1.** *A syntactically correct closed system $\mathcal{S} = (\mathcal{P}, \mathcal{C}, map)$ with a set $\mathcal{P}$ of correct patterns $P_1, \ldots P_n$ and a set $\mathcal{C}$ of correct components $C_1, \ldots, C_m$ is semantically correct.*

*Proof.* 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: $M_1 \| \ldots \| M_k \| M_i^P \models \phi_i \wedge \neg\delta$. As all $M_j^G := M_1 \| \ldots \| M_k \| M_j^P$ have disjoint signal and labelling sets and no label used in $\phi_i$ is in the labels set of the other models ($\mathcal{L}(M_j^G) \cap \mathcal{L}(\phi_i) = \emptyset$) we can combine them for all $j \in [1, n] - \{i\}$ in parallel while preserving $\phi_i$ and have: $M_1^G \| \ldots \| M_n^G \models \phi_i \wedge \neg\delta$. Let $M_j^B$ be the parallel composition of all role automata relation to component $C_j$, then holds $M_j^C \sqsubseteq_{I/O} M_j^B$ for all correct components $C_i$. By replacing $M_j^B$ by $M_j^C$ in an appropriate reordered term for $M_1^G \| \ldots \| M_n^G \models \phi_i \wedge \neg\delta$ we thus can due to Lemma 3 conclude $M_1^P \| \ldots \| M_n^P \| M_1^C \| \ldots \| M_m^C \models \phi_i \wedge \neg\delta$. Condition 5 for a semantically correct system can thus be obtained simply by using the above derivation for all $i \in [1, n]$.
To proof condition 6 we can simply derive for any $i$ from $M_i^C$ a correct component that $M_i^C \models \psi_i^C$ holds. Invariants are compositional and therefore we have $M_1^P \| \ldots \| M_n^P \| M_1^C \| \ldots \| M_m^C \models \psi_i^C$ due to condition 2 of Definition 3 and the above result of deadlock freedom. By iteration over all $i \in [1, m]$ we thus can also obtain condition 6. □

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 addressed using the protocol statecharts.

## 7.3 Refining Pattern Roles

In general a valid transformation that ensures our notion of refinement ($\sqsubseteq$) also has to preserve deadlock freedom. As we require not only that $M_i^r$ refines $map(M_i^r)$ but that $M^C$ refines $M_i^r$. This is a rather hard problem as we have to take into account all other port automata and $M^s$. We can however avoid this problem by assuming that the resulting $M^C$ will be checked for deadlocks. Thus we require only a transformation that ensures refinement if no deadlock occurs. We can therefore restrict our attention to the task of deriving $M_i^r$ from $M_i$ using a set of transformation rules which later ensure either for all $j \in [1, h]$ that $M_1^r \| \ldots \| M_h^r \| M_i^s \sqsubseteq_{I/O} M_j$ holds or $M_1^r \| \ldots \| M_h^r \| M_i^s \models \delta$. We name all such transformations to be *valid* ones (cf. [14]).

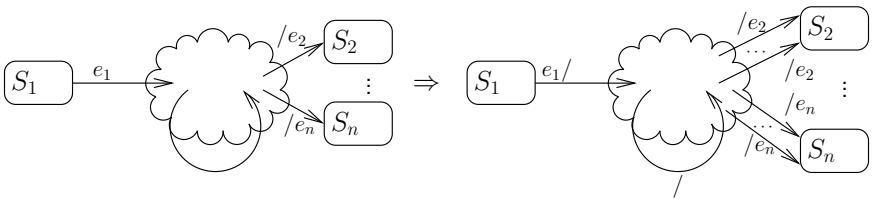


Figure 17: A transformation which preserves refinement

In Figure 17 one possible valid transformation is presented. We use $e_i$ to denote externally visible signals relevant for the coordination within the pattern. 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 only be connected by internal transitions and the resulting automaton will nondeterministically 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 $M^s$ by sending and receiving any internal signals denoted by 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. The added signals can be employed to coordinate the required or possible decisions with $M^s$ as required.

# 8 Conclusion and Future Work

The presented approach is based on a restricted notion of patterns suitable for the considered domain of mechatronic systems. These patterns further enable to derive the required component behavior by means of refinement steps for each role. Finally, the required overall system can be built by only composing compo-

nents 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 first verified in ”isolation” using existing real-time model checking tools. The overall correctness can be derived by only ensuring the syntactically correct composition of the system.

As the presented approach only employs standard UML modeling and specification techniques with minor domain-specific extensions, the developer is not confronted with non-standard software engineering notations such as temporal logics.

The possible clear separation between pattern and component design also enables the evolution of the system. The patterns form a contract between a component and its environment which permits to exchange a given component by another one that also fulfills the contract. Another option also studied within the mentioned case study is that the components themselves are allowed to adjust their run-time behavior as long as they stay within the given contract. Thus, component behavior which includes self-adaptation and self-optimization becomes possible.

Similar to the DistanceCoordination pattern we investigate other domain specific patterns which are collected in a library. This brings significant ease in system design by reusing domain specific design solutions.

While in the current state we can only report our experiences with the proposed sequence of design steps and the employed tools, we plan to further provide tool support for all presented activities within the Fujaba UML CASE tool [16]. This includes automating the translation process and integrating the RAVEN model checker.

**Acknowledgements**
We thank Björn Axenath, Ekkart Kindler, and Yuhong Zhao for their comments on earlier versions of the technical report.

# References

[1] R. Alur, C. Courcoubetis, and D.L. Dill. Model Checking for Real-Time Systems. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 414–425, Washington, D.C., 1990.

[2] Maher Awad, Juha Kuusela, and Jurgen Ziegler. *Object-Oriented Technology for Real-Time Systems: A Practical Approach Using OMT and Fusion*. Prentice Hall, 1996.

[3] Gerd Behrmann, Johan Bengtsson, Alexandre David, Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL implementation secrets. In *Proc. of 7*th *International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems*, 2002.

[4] J. Bradfield, J. Kuester Filipe, and P. Stevens. Enriching OCL Using Observational mu-Calculus. In R.-D. Kutsche and H. Weber, editors, *Fundamental Approaches to Software Engineering (FASE 2002), Grenoble, France*, volume 2306 of *LNCS*. Springer, April 2002.

[5] S. Campos, E.M. Clarke, and M. Minea. The Verus Tool: A Quantitative Approach to the Formal Verification of Real-Time Systems. In *Conference on Computer Aided Verification (CAV)*, volume 1254 of *LNCS*, pages 452–455. Springer, June 1997.

[6] M.V. Cengarle and A. Knapp. Towards OCL/RT. In L.-H. Eriksson and P.A. Lindsay, editors, *Formal Methods – Getting IT Right, International Symposium of Formal Methods Europe, Copenhagen, Denmark*, volume 2391 of *LNCS*, pages 389–408. Springer, 2002.

[7] William Chan, Richard J. Anderson, Paul Beame, Steve Burns, Francesmary Modugno, David Notkin, and Jon D. Reese. Model Checking Large Software Specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, 1998.

[8] E. M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, January 2000.

[9] Bruce Powel Douglass. *Real-Time UML: Developing Efficient Objects for Embedded Systems*. The Addison-Wesley Object Technology Series. Addison-Wesley, October 1999. Second Edition.

[10] E. Emerson, A. Mok, A. Sistla, and J. Srinivasan. Quantitative Temporal Reasoning. *Journal of Real-Time Systems*, 4(4):331–352, 1992.

[11] Stephan Flake and Wolfgang Mueller. An OCL Extension for Real-Time Constraints. In *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*, volume 2263 of *LNCS*, pages 150–171. Springer, February 2002.

[12] H. Giese and S. Burmester. Real-Time Statechart Semantics. Technical Report tr-ri-03-239, Computer Science Department, University of Paderborn, June 2003.

[13] Holger Giese. Contract-based Component System Design. In Jr. Ralph H. Sprague, editor, *Thirty-Third Annual Hawaii International Conference on System Sciences (HICSS-33), Maui, Hawaii, USA*. IEEE Computer Press, January 2000.

[14] Holger Giese. A formal calculus for the compositional pattern-based design of correct real-time systems. Technical Report tr-ri-03-240, Computer Science Department, University of Paderborn, July 2003.

[15] Hassan Gomaa. *Designing Concurrent, Distributed, and Real-Time Applications with UML*. Addison-Wesley, January 2000.

[16] H.J. Köhler, U. Nickel, J. Niere, and A. Zündorf. Integrating UML Diagrams for Production Control Systems. In *Proc. of the* $22^{nd}$ *International Conference on Software Engineering (ICSE), Limerick, Irland*, pages 241–251. ACM Press, 2000.

[17] Harry Li, Shriram Krishnamurthi, and Kathi Fisler. Verifying Cross-Cutting Features as Open Systems. In William G. Griswold, editor, *Proceedings of the Tenth ACM SIGSOFT Symposium on the Foundations of Softare Engineering (FSE-10)*, Charleston, South Carolina, USA, November 2002. ACM Press.

[18] Gerald Lüttgen, Michael von der Beeck, and Rance Cleaveland. A Compositional Approach to Statecharts Semantics. In *Proceedings of the Eighth International Symposium on Foundations of Software Engineering for Twenty-first Century Applications, November 2000, San Diego, CA USA*, pages 120–129, 2000.

[19] J. Misra and M. Chandy. Proofs of Networks of Processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, 1981.

[20] Object Management Group. UML Profile for Schedulability, Performance, and Time Specification. OMG Document ptc/02-03-02, September 2002. URL: http://cgi.omg.org/docs/ptc/02-03-02.pdf.

[21] Object Management Group. UML Superstructure Submission V2.0. OMG Document ad/03-04-01, April 2003. URL: http://www.omg.org/cgi-bin/doc?ad/2003-04-01.

[22] Jürgen Ruf. RAVEN: Real-Time Analyzing and Verification Environment. *Journal on Universal Computer Science (J.UCS), Springer*, 7(1):89–104, February 2001.

[23] Jürgen Ruf and Thomas Kropf. Analyzing Real-Time Systems. In *Design, Automation and Test in Europe (DATE), Paris, France*. IEEE Computer Society Press, March 2000.

[24] Bran Selic, Garth Gullekson, and Paul Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, Inc., 1994.

[25] Bran Selic and Jim Rumbaugh. Using UML for Modeling Complex Real-Time Systems. Techreport, ObjectTime Limited, 1998.

[26] S. Yovine. Kronos: A verification tool for real-time systems. *International Journal of Software Tools for Technology Transfer*, 1:123–133, October 1997.

# A Model Checking Results

During the previous sections we described how to build a system with components and patterns. Within this section we give the results obtained when model checking this system. We will report how much time and memory capacity has been required to check the constraints for the different models. Additionally, we will report the number of states and transitions for each model that has been build.

In Section A.1 we briefly describe the explored configuration options of the model checker RAVEN. Afterwards, we give in Section A.2 an overview of checking the distance coordination pattern. Section A.3 shows the verification of a single shuttle component. In section A.4 we built a system consisting of several shuttles to crosscheck our results and to show the improvement in model checking achieved by our approach.

## A.1 Options for Model Checking with RAVEN

As stated in Section 2 we used the RAVEN model checker to verify the models. Therefore the model has to get translated into the RAVEN input language (ril) which is an interval structure. This has been done manually.

After loading a model in ril format into the model checker the user has the ability to adjust RAVEN to his requirements. This means he can decide how the model should be presented internally and how this internal representation should be build. At first, there is the possibility to choose MTBDD (Multi Terminal Binary Decision Diagram) or ROBDD (Reduced Ordered Binary Decision Diagram) as internal representing of the model. This is done by selecting *multiple delay* or resp. *unit delay*. The following algorithms are offered by RAVEN for building a model in one or both internal representations:

- Standard: Standard expansion-composition-reduction algorithm for small and medium sized systems in multiple delay mode.
- Prerestriction: Computes the set of reachable states after composition and intersects the transition realtion with it. Works well for small and medium sized systems in unit delay mode. In multiple delay it works only if the initial set is chosen as start set.
- Combined: Performs composition and reduction in one step. Is the preferred algorithm for large systems in multiple delay mode. Cannot be used in the unit delay mode because reduction is not possible in this mode.
- Restriction: Computes the set of reachable states after composition and restricts the set of states in every traversal step in the checking algorithm by this set.

- Incremental: Builds the composition structure incrementally by traversing the original structures. The resulting structure contains only reachable states and transitions. Usefull in cases of large systems in unit delay mode.

Additionally RAVEN offers two optimization and two minimization techniques. The optimization techniques are

- Prediction: Uses local timing informations of the modules to find times in which no state change happens. This may increase the time progress for composition.

- Partition: Partitions the transitions into relations such that the structure is stored in an array of ROBDDs. This often leads to faster composition but increased memory consumption.

And these are the minimization techniques

- Minpath: Builds equivalence classes of states which behave same and introduces new transitions. This may lead to a reduced number of MTBDD nodes.

- Reencode: Reencodes new introduced encoding variables.

For defining the initial set in multiple delay mode RAVEN offers the following three options

- Init: Starts in the initial states defined in the INIT part within the ril file.

- Iprod: Sets the start set to the product set of all original interval structure sets.

- One: Sets the start set to the product of at least one interval structure state and all possible stutter states.

We checked our models with multiple delay as well as with unit delay and then changed the algorithms and minimization techniques. Since it is possible to choose both optimization techniques at a time we decided to do all checks with both techniques activated. As the initial set of states is specified in our models we chose *init* in the appropriate adjustment. Table 1 lists the different configurations.

When the model is internally represented in unit delay mode only the configurations 1,2,4 and 5 are possible.

All test were performed on an Intel Pentium4 with 2.40 GHz CPU, 512 KB cache size and Red Head Linux 7.3 as operating system.

| Configuration | Algorithm | Minimization |
|---|---|---|
| 1 | Standard | Minpath |
| 2 | Prerestriction | Minpath |
| 3 | Combined | Minpath |
| 4 | Restriction | Minpath |
| 5 | Incremental | Minpath |
| 6 | Standard | Reencode |
| 7 | Prerestriction | Reencode |
| 8 | Combined | Reencode |
| 9 | Restriction | Reencode |
| 10 | Incremental | Reencode |
| 11 | Standard | None |
| 12 | Prerestriction | None |
| 13 | Combined | None |
| 14 | Restriction | None |
| 15 | Incremental | None |

Table 1: List of all used RAVEN configurations

## A.2 Checking the Distance Coordination Pattern

In this section we describe how the model for the distance coordination pattern was build and checked and also list the model checking results.

### A.2.1 Building the Model

To check the distance coordination pattern one has to translate the statecharts given in section 5.1 into the RAVEN input language.

As there was a mistake in the first model, see Section 6.1, we corrected the model and checked it again. We will refere to these models as the *firstModel* and the *correctedModel*.

The following tables show the size of the resulting systems by listing the number of BDD nodes, states and transitions for the unit delay mode, shown in table 3 as well as for the multiple delay mode in Table 2. The asterisk (*) within these tables denotes that the command to count the number of states and transitions leads to no response.

### A.2.2 Compositional Constraints

The required property of the distance coordination pattern described in Section 5.1 is:

| | FirstModel | CorrectedModel |
|---|---|---|
| Nodes after minimization | 12.596 | 13.789 |
| Nodes before minimization | 12.775 | 13.815 |
| Reachable states | 1.720.586.736 | 2.027.649.280 |
| Main states | 1.704.144.768 | 2.008.838.016 |
| Reachable transitions | 2.249.306.112 | 3.054.612.480 |
| Main transitions | 2.249.306.112 | 3.054.612.480 |

Table 2: Number of BDD nodes, states and transitions for both versions of the distance coordination pattern model with multiple delay as internal representation

| | 1 | 2 | 4 | 5 |
|---|---|---|---|---|
| BDD nodes | 16.592 | 10.754 | 16.592 | 10.745 |
| States | * | 343437 | * | * |
| Transitions | * | 980495 | * | * |

Table 3: Number of BDD nodes, states and transitions for the first model of the distance coordination pattern model with unit delay as internal representation

| | 1 | 2 | 4 | 5 |
|---|---|---|---|---|
| BDD nodes | 18.082 | 11.740 | 18.082 | 11.740 |
| States | * | * | * | * |
| Transitions | * | * | * | * |

Table 4: Number of BDD nodes, states and transitions in for the second model of the distance coordination pattern model with unit delay as internal representation

```
context DistanceCoordination inv:
     not (self.oclInState(RearRole::Main::convoy) and
        self.oclInState(FrontRole::Main::noConvoy))
   and
     (self.oclInState(FrontRole::Ping::sendData)
     implies
       self@post(1,6)->forAll(p:OclPath |
              p->exists(c:OclConfiguration |
              c->includes(RearRole::Pong::incoming)|
              c->includes(Channel::fail))))
```

For model checking with RAVEN it is divided into the following two properties given in ACCTL:

- DistCoord1: This constraint describes that at no time the rear shuttle is in state *convoy* while the front shuttle is in state *noConvoy*.

  ```
  DistCoord1 := AG !(rearRoleMain.convoy & frontRoleMain.noConvoy)
  ```

  We further use **DistCoord1** to refer to this constraint.

- DistCoord2: It describes that not later than 6 time steps after the front shuttle sent some data the rear shuttle has to receive this data or the channel which forwards the data has to be in state error.[4]

  ```
  DistCoord2 := AG((frontRolePing.state = frontRolePing.sendData) ->
                    AF[1 ,6](channel1.inError |
                             (rearRolePong.state = rearRolePong.incoming)))
  ```

  We further use **DistCoord2** to refer to the time required to check this constraint.

### A.2.3 Additional Constraints

To check that the manually derived models are correct, we additional check the following properties:

- AddCheck1: States that the front shuttle has repeatedly to send data to the rear shuttle. At most 10 time steps may pass after one sending is done and the next sending has to be performed. This is stated in the following constraint:

  ```
  AddCheck1 := AG AF [1, 10](frontRolePing.state =
                                          frontRolePing.sendData)
  ```

[4]This formula is still an ACCTL formula, because $p \rightarrow q = ! \, p \mid q$ and the concrete value `(frontRolePing.state = frontRolePing.sendData)` of $p$ in this formula is after negation still in ACCTL.

- AddCheck2: Ensures that there is at least one way through the state space at which the environment does not cause a channel to enter the error state. Although the pattern is a closed system, we introduced the environment to non-deterministically cause a channel to switch into the error state.

```
AddCheck2 := EG !(environment.error)
```

- Emergency: Requires that in a schedule of at most 16 time steps the rear shuttle has to be in an emergency state or in a state that allows the shuttle to receive data.

```
Emergency := AG AF [0, 16]((rearRolePong.state = rearRolePong.incoming)
                  |(rearRolePong.state = rearRolePong.emergency))
```

**AddCheck1**, **AddCheck2** and **Emergency** are further used to refer to the time requiered to check the related property.

Our approach demands that the models have to be deadlock free. This additional check is automatically performed by RAVEN.

### A.2.4 Erroneous Version

In Tabel 5 to Table 7 the results for checking the first model with multiple delay mode are listed. Table 8 shows the results for checking this model in unit delay mode. Within this tables time is given in seconds whereas memory is given in kilobyte. The line labeled with "time" contains the time needed to perform a complete check, which means parsing the ril file, building the internal representation, perform the minimization and model check the model (MC).

### A.2.5 Corrected Version

The tables in Section A.2.4 list results for checking the erroneous model of the distance coordination pattern whereas the following tables will show the results for the corrected model. Table 9 to 11 contain the results for the corrected model in multiple delay and Table 12 the results for this model in unit delay mode. As in the Section above time is give in seconds and memory in kilobyte.

## A.3 Checking the Shuttle Component

After verifiying the pattern, we built a component that refined the roles described in the pattern and checked whether this component is correct.

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Time [sec] | 14,54 | 17,02 | 15,03 | 17,24 | 20,99 |
| Parsing [sec] | 0,04 | 0,05 | 0,03 | 0,04 | 0,05 |
| Composition [sec] | 13,62 | 16,09 | 14,2 | 16,27 | 20,05 |
| Minimization [sec] | 0,27 | 0,29 | 0,26 | 0,32 | 0,27 |
| MC [sec] | 0,55 | 0,55 | 0,51 | 0,56 | 0,57 |
| DistCoord1 [sec] | 0,05 | 0,05 | 0,05 | 0,05 | 0,05 |
| DistCoord2 [sec] | 0,02 | 0,01 | 0,02 | 0,02 | 0,02 |
| AddCheck1 [sec] | 0,18 | 0,16 | 0,16 | 0,16 | 0,15 |
| AddCheck2 [sec] | 0,01 | 0,01 | 0,01 | 0,01 | 0,01 |
| Emergency [sec] | 0,19 | 0,32 | 0,27 | 0,32 | 0,34 |
| Memory [kbyte] | 7.100 | 6.044 | 6.196 | 6.816 | 5.984 |

Table 5: Time in msec and memory in kB needed to parse, composite and check the first model with configuration 1 to 5. For the internal representation multiple delay is used.

| | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|
| Time [sec] | 14,22 | 17,09 | 14,98 | 17,16 | 20,85 |
| Parsing [sec] | 0,05 | 0,05 | 0,04 | 0,05 | 0,04 |
| Composition [sec] | 13,34 | 16,18 | 14,18 | 16,28 | 19,93 |
| Minimization [sec] | 0,28 | 0,27 | 0,24 | 0,32 | 0,27 |
| MC [sec] | 0,51 | 0,55 | 0,48 | 0,57 | 0,58 |
| DistCoord1 [sec] | 0,04 | 0,05 | 0,05 | 0,06 | 0,05 |
| DistCoord2 [sec] | 0,02 | 0,02 | 0,01 | 0,02 | 0,03 |
| AddCheck1 [sec] | 0,16 | 0,16 | 0,14 | 0,16 | 0,14 |
| AddCheck2 [sec] | 0,01 | 0,01 | 0,01 | 0,01 | 0,02 |
| Emergency [sec] | 0,28 | 0,31 | 0,27 | 0,32 | 0,34 |
| Memory [kbyte] | 7.100 | 6.044 | 6.196 | 6.816 | 5.984 |

Table 6: Time in msec and memory in kB needed to parse, composite and check the first model with configuration 6 to 10. For the internal representation multiple delay is used.

| | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|
| Time [sec] | 14,1 | 16,99 | 15,09 | 17,12 | 21,01 |
| Parsing [sec] | 0,05 | 0,05 | 0,05 | 0,04 | 0,05 |
| Composition [sec] | 13,23 | 16,09 | 14,27 | 16,17 | 20,06 |
| Minimization [sec] | 0,16 | 0,28 | 0,26 | 0,32 | 0,27 |
| MC [sec] | 0,51 | 0,54 | 0,48 | 0,54 | 0,6 |
| DistCoord1 [sec] | 0,05 | 0,05 | 0,05 | 0,05 | 0,05 |
| DistCoord2 [sec] | 0,02 | 0,02 | 0,02 | 0,02 | 0,02 |
| AddCheck1 [sec] | 0,15 | 0,16 | 0,15 | 0,14 | 0,16 |
| AddCheck2 [sec] | 0,01 | 0,01 | 0,01 | 0,01 | 0,02 |
| Emergency [sec] | 0,28 | 0,3 | 0,25 | 0,32 | 0,35 |
| Memory [kbyte] | 7.100 | 6.044 | 6.196 | 6.816 | 5.984 |

Table 7: Time in msec and memory in kB needed to parse, composite and check the first model with configuration 11 to 15. For the internal representation multiple delay is used.

| | 1 | 2 | 4 | 5 |
|---|---|---|---|---|
| Time [sec] | 1,59 | 2,74 | 3,41 | 6,17 |
| Parsing [sec] | 0,05 | 0,05 | 0,05 | 0,04 |
| Composition [sec] | 0,08 | 2,18 | 1,96 | 5,69 |
| MC [sec] | 1,45 | 0,49 | 1,38 | 0,43 |
| DistCoord1 [sec] | 0,54 | 0,04 | 0,25 | 0,04 |
| DistCoord2 [sec] | 0,51 | 0,03 | 0,08 | 0,03 |
| AddCheck1 [sec] | 0,14 | 0,17 | 0,5 | 0,13 |
| AddCheck2 [sec] | 0,02 | 0,02 | 0,04 | 0,02 |
| Emergency [sec] | 0,24 | 0,23 | 0,51 | 0,21 |
| Memory [kbyte] | 4.544 | 4.800 | 5.248 | 4.800 |

Table 8: Time in msec and memory in kB needed to parse, composite and check the first model with configuration 1, 2, 4 and 5. The other configurations are not possible because for the internal representation unit delay is used.

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Time [sec] | 14,43 | 17,51 | 15,56 | 18,14 | 21,51 |
| Parsing [sec] | 0,05 | 0,05 | 0,04 | 0,05 | 0,05 |
| Comp [sec] | 13,52 | 16,65 | 14,78 | 17,12 | 20,57 |
| Minimization [sec] | 0,31 | 0,3 | 0,29 | 0,33 | 0,31 |
| MC [sec] | 0,5 | 0,58 | 0,5 | 0,59 | 0,54 |
| DistCoord1 [sec] | 0,01 | 0,02 | 0,17 | 0,02 | 0,01 |
| DistCoord2 [sec] | 0,03 | 0,03 | 0,03 | 0,03 | 0,02 |
| AddCheck1 [sec] | 0,16 | 0,19 | 0,17 | 0,17 | 0,17 |
| AddCheck2 [sec] | 0,01 | 0,01 | 0,02 | 0,01 | 0,02 |
| Emergency [sec] | 0,29 | 0,33 | 0,27 | 0,36 | 0,32 |
| Memory [kbyte] | 7.420 | 6.172 | 6.272 | 7.068 | 6.112 |

Table 9: Time in msec and memory in kB needed to parse, composite and check the corrected pattern with configuration 1 to 5. For the internal representation multiple delay is used.

| | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|
| Time [sec] | 14,61 | 17,6 | 16,06 | 17,87 | 21,04 |
| Parsing [sec] | 0,05 | 0,04 | 0,05 | 0,05 | 0,04 |
| Composion [sec] | 13,69 | 16,65 | 15,17 | 16,88 | 20,11 |
| Minimization [sec] | 0,31 | 0,31 | 0,29 | 0,35 | 0,3 |
| MC [sec] | 0,51 | 0,56 | 0,49 | 0,55 | 0,55 |
| DistCoord1 [sec] | 0,01 | 0,02 | 0,01 | 0,02 | 0,01 |
| DistCoord2 [sec] | 0,03 | 0,03 | 0,02 | 0,03 | 0,02 |
| AddCheck1 [sec] | 0,16 | 0,17 | 0,16 | 0,16 | 0,17 |
| AddCheck2 [sec] | 0,01 | 0,01 | 0,02 | 0,01 | 0,02 |
| Emergency [sec] | 0,03 | 0,33 | 0,28 | 0,33 | 0,33 |
| Memory [kbyte] | 7.420 | 6.172 | 6.272 | 7.068 | 6.112 |

Table 10: Time in msec and memory in kB needed to parse, composite and check the corrected pattern with configuration 6 to 10. For the internal representation multiple delay is used.

| | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|
| Time [sec] | 14,49 | 17,35 | 16,11 | 17,99 | 21,35 |
| Parsing [sec] | 0,04 | 0,04 | 0,05 | 0,05 | 0,05 |
| Composition [sec] | 13,63 | 16,37 | 15,22 | 17 | 20,39 |
| Minimization [sec] | 0,3 | 0,3 | 0,29 | 0,36 | 0,3 |
| MC [sec] | 0,47 | 0,6 | 0,51 | 0,54 | 0,57 |
| DistCoord1 [sec] | 0,01 | 0,02 | 0,01 | 0,01 | 0,01 |
| DistCoord2 [sec] | 0,02 | 0,03 | 0,03 | 0,02 | 0,03 |
| AddCheck1 [sec] | 0,15 | 0,17 | 0,17 | 0,16 | 0,18 |
| AddCheck2 [sec] | 0,01 | 0,02 | 0,02 | 0,01 | 0,01 |
| Emergency [sec] | 0,28 | 0,36 | 0,28 | 0,34 | 0,34 |
| Memory [kbyte] | 7.420 | 6.172 | 6.272 | 7.068 | 6.112 |

Table 11: Time in msec and memory in kB needed to parse, composite and check the corrected pattern with configuration 11 to 15. For the internal representation multiple delay is used.

| | 1 | 2 | 4 | 5 |
|---|---|---|---|---|
| Time [sec] | 1,21 | 3,06 | 3,94 | 6,76 |
| Parsing [sec] | 0,05 | 0,04 | 0,05 | 0,05 |
| Composition [sec] | 0,08 | 2,5 | 2,46 | 6,22 |
| MC [sec] | 1,07 | 0,5 | 1,41 | 0,47 |
| DistCoord1 [sec] | 0,12 | 0,01 | 0,01 | 0,01 |
| DistCoord2 [sec] | 0,52 | 0,04 | 0,08 | 0,04 |
| AddCheck1 [sec] | 0,14 | 0,16 | 0,58 | 0,18 |
| AddCheck2 [sec] | 0,2 | 0,02 | 0,05 | 0,01 |
| Emergency [sec] | 0,27 | 0,28 | 0,69 | 0,24 |
| Memory [kbyte] | 4.540 | 4.928 | 5.276 | 4.800 |

Table 12: Time in msec and memory in kB needed to parse, composite and check the corrected pattern with configuration 1, 2, 4 and 5. The other configurations are not possible because for the internal representation unit delay is used.

| | Shuttle Component |
|---|---|
| Nodes after minimization | 835 |
| Nodes before minimization | 873 |
| Reachable states | 18.770 |
| Main states | 18.626 |
| Reachable transitions | 610.056 |
| Main transitions | 610.056 |

Table 13: Number of BDD node, states and transitions for the shuttle component with multiple delay as internal representation

| | 1 | 2 | 4 | 5 |
|---|---|---|---|---|
| BDD nodes | 1.452 | 265 | 1.452 | 265 |
| States | 50.122.800 | 5.822 | 5.822 | 5.822 |
| Transitions | 741.323.520 | 38.112 | 741.323.520 | 38.112 |

Table 14: Number of BDD node, states and transitions for the shuttle component with unit delay as internal representation

### A.3.1 Building the Model

A component consists of refined pattern roles and a synchonization statechart. This is not a closed system, because the roles expect input signals comming via the channels, but these channels are only modeled within the pattern. Therefore the unconnected transitions are replaced by non-deterministic ones when building the RAVEN model of a shuttle component. The size of the resulting model is shown in Tables 13 and 14. Table 13 shows the size in case of multiple delay mode and Table 14, the case of unit delay as internal representation.

### A.3.2 Compositional Constraints

Each shuttle participates in two different instances of the distance coordination pattern. Due to this fact each shuttle has one port realizing a front role and one port realizing a rear role. The synchronization of this two roles has to fulfill the in Section 5.2 described combination of the local invariants:

```
context Shuttle inv:
    frontRole.oclInState(convoy) implies
      not self.CanBrakeFully
  and
    rearRole.oclInState(convoy) implies
      self.CanBrakeFully
```

It is mapped to the following CCTL property with additional existential quantification

```
AG (EXISTS CanBrakeFully:
   (frontRoleMain.convoy -> ! CanBrakeFully) and
   (rearRoleMain.convoy -> CanBrakeFully))
```

This formula is valid for all state combinations besides both roles are in convoy and thus we only have to check:[5]

```
ShuttleInv := AG !(rearRoleMain.convoy & frontRoleMain.convoy)
```

This constraint describes that the rear role and front role of a shuttle are not allowed to be in state convoy at the same time. **ShuttleInv** is further used to refer to the time needed to check the shuttle invariant.

### A.3.3 Additional Constraints

When checking a shuttle component we checked the additional constraints AddCheck1 and Emergency, which are the same we used during the verification of the distance coordination pattern. AddCheck2 concerns the environment variable introduced to model that channels non-deterministically change into the emergency state. Because channels are not considered within the shuttle component this property cannot be checked.

### A.3.4 Check

In the following we list the results obtained when checking the component. The tables are given for multiple delay and for unit delay. Table 15 to 17 contain the results for checking the shuttle component in multiple delay mode whereas Table 18 contains the results for checking the model in unit delay mode. Times noted with 0,00 means that this corresponding process needed less than 0,01 seconds[6].

## A.4 Checking Some Shuttle System Configurations

To show the improvement that can be achived by using our approach we build systems consisting of several shuttles and also checked them with RAVEN.

[5] In an automatic mapping this required mapping step for invariants can be easily done using for instance an appropriate ROBDD package which supports quantification over its finite domain variables.

[6] These are the values RAVEN presents the user

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Time [sec] | 0,09 | 0,09 | 0,07 | 0,08 | 0,09 |
| Parsing [sec] | 0,01 | 0,02 | 0,01 | 0,01 | 0,02 |
| Composition [sec] | 0,05 | 0,05 | 0,05 | 0,06 | 0,05 |
| Minimization [sec] | 0,00 | 0,00 | 0,00 | 0,00 | 0,00 |
| MC [sec] | 0,03 | 0,02 | 0,01 | 0,01 | 0,02 |
| ShuttleInv [sec] | 0,00 | 0,00 | 0,00 | 0,00 | 0,00 |
| AddCheck2 [sec] | 0,01 | 0,00 | 0,00 | 0,00 | 0,00 |
| Emergency [sec] | 0,02 | 0,02 | 0,01 | 0,01 | 0,02 |
| Memory [kbyte] | 3.420 | 3.420 | 3.416 | 3.452 | 3.420 |

Table 15: Time in msec and memory in kB needed to parse, composite and check the shuttle component with configuration 1 to 5. For the internal representation multiple delay is used.

| | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|
| Time [sec] | 0,09 | 0,08 | 0,08 | 0,11 | 0,08 |
| Parsing [sec] | 0,01 | 0,01 | 0,01 | 0,02 | 0,01 |
| Composition [sec] | 0,05 | 0,05 | 0,04 | 0,06 | 0,06 |
| Minimization [sec] | 0,01 | 0,00 | 0,00 | 0,01 | 0,00 |
| MC [sec] | 0,02 | 0,02 | 0,02 | 0,02 | 0,01 |
| ShuttleInv [sec] | 0,00 | 0,00 | 0,00 | 0,00 | 0,00 |
| AddCheck2 [sec] | 0,00 | 0,00 | 0,00 | 0,01 | 0,00 |
| Emergency [sec] | 0,02 | 0,02 | 0,02 | 0,01 | 0,01 |
| Memory [kbyte] | 3.420 | 3.420 | 3.416 | 3.452 | 3.420 |

Table 16: Time in msec and memory in kB needed to parse, composite and check the corrected and refined pattern with configuration 6 to 10. For the internal representation multiple delay is used.

| | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|
| Time [sec] | 0,09 | 0,09 | 0,08 | 0,1 | 0,09 |
| Parsing [sec] | 0,01 | 0,02 | 0,01 | 0,01 | 0,02 |
| Composition [sec] | 0,05 | 0,05 | 0,05 | 0,05 | 0,05 |
| Minimization [sec] | 0,00 | 0,00 | 0,00 | 0,00 | 0,01 |
| MC [sec] | 0,03 | 0,01 | 0,01 | 0,03 | 0,01 |
| ShuttleInv [sec] | 0,00 | 0,00 | 0,00 | 0,00 | 0,00 |
| AddCheck2 [sec] | 0,01 | 0,00 | 0,00 | 0,01 | 0,00 |
| Emergency [sec] | 0,02 | 0,01 | 0,01 | 0,02 | 0,01 |
| Memory [kbyte] | 3.420 | 3.420 | 3.416 | 3.452 | 3.420 |

Table 17: Time in msec and memory in kB needed to parse, composite and check the corrcted and refined pattern with configuration 11 to 15. For the internal representation multiple delay is used.

| | 1 | 2 | 4 | 5 |
|---|---|---|---|---|
| Time [sec] | 0,07 | 0,03 | 0,04 | 0,04 |
| Parsing [sec] | 0,01 | 0,02 | 0,00 | 0,01 |
| Composition [sec] | 0,01 | 0,01 | 0,02 | 0,03 |
| MC [sec] | 0,05 | 0,00 | 0,02 | 0,00 |
| ShuttleInv [sec] | 0,02 | 0,00 | 0,00 | 0,00 |
| AddCheck1 [sec] | 0,01 | 0,00 | 0,01 | 0,00 |
| Emergency [sec] | 0,02 | 0,00 | 0,01 | 0,00 |
| Memory [kbyte] | 3.372 | 3.344 | 3.340 | 3.344 |

Table 18: Time in msec and memory in kB needed to parse, composite and check the shuttle component with configuration 1, 2, 4 and 5. For the internal representation unit delay is used.

### A.4.1 Building the Model

To build a system consisting of several shuttles we took the shuttle component from above. Within this component we used non-determinism to be able to check the component without modelling the environment. When modelling a complete convoy this environment is needed. Therefor we modified the component slightly by replacing the non-determinism by determinism and thus modeled the environment. Afterwards we multiplied this component. The components got connected via their ports and additional channels to build one convoy. This connection was done like in the case of the front and the rear role of the distance coordination pattern.

### A.4.2 Compositional Constraints

In this case we checked whether there is a minimal space between two shuttles of the convoy at any time. Because the shuttle components and their ports are copies from the above components, the ports are named in the same way as above and we can use the distCoord1 and distCoord2 properties to check the correctness of the system. To be able to compare the results of this checking with those of the compositional approach we checked this formulas only for two shuttles instead of the whole convoy.

Within this system we have no patterns so it is necessary to adjust the properties slightly. The two distance properties now look like:

```
DistCoord1 := AG !(rearRoleMain_1.convoy & frontRoleMain_2.convoy)
DistCoord2 := AG ((frontRolePing_2.state = frontRolePing_2.sendData)
              -> AF[1, 6](channel.in_error
                        |(lastRolePong_1.state = lastRolePong_1.incoming)))
```

### A.4.3 Additional Constraints

When checking a shuttle component we checked the additional constraints AddCheck1, AddCheck2 and Emergency, which are the same we used during the verification of the distance coordination pattern.

### A.4.4 Check

Figure 18 shows the time needed to check a convoy with a changing number of shuttles. Figure 19 shows the number of BDD nodes generated for the models and Figure 20 shows the needed memory to perform this checks. These checks were performed in multiple delay mode only.

The largest convoy tested is the one consisting of 25 shuttles. When checking convoys with 30 or more shuttles RAVEN failed with no error message.

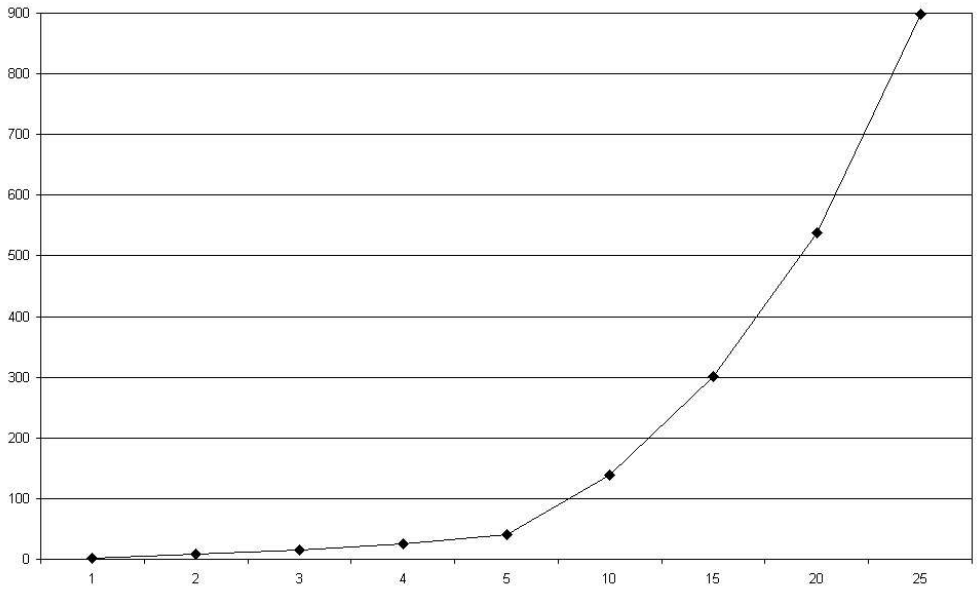


Figure 18: Time in seconds needed to check convoys with 1 to 25 shuttles

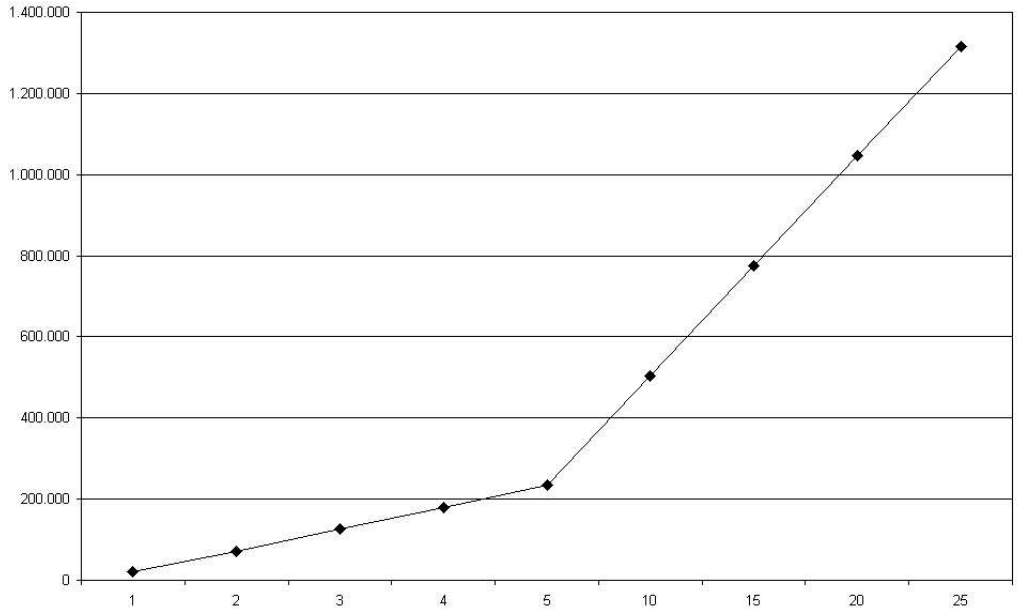


Figure 19: Number of BDD nodes created when checking convoys with 1 to 25 shuttles

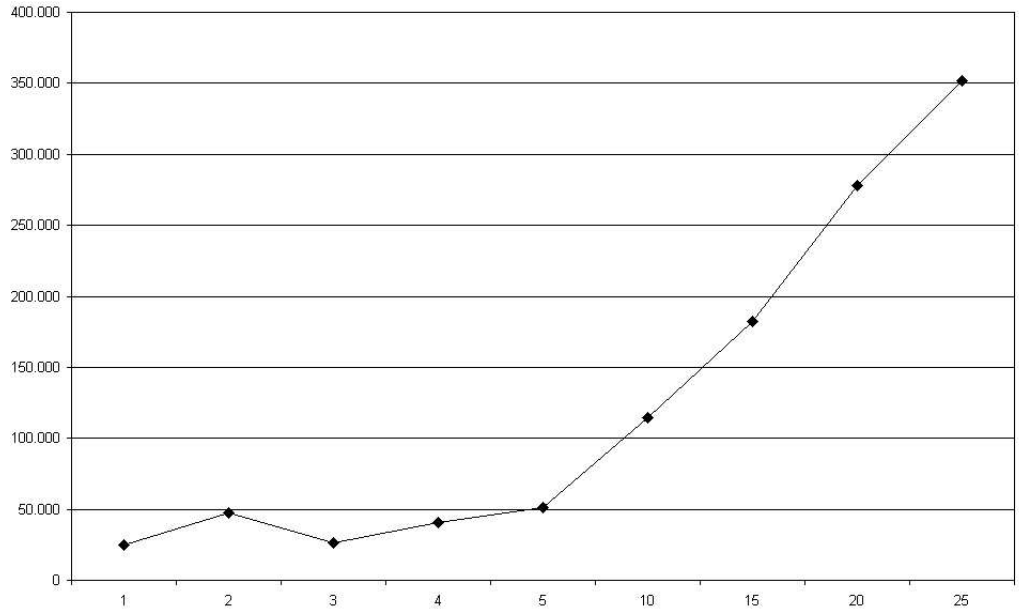


Figure 20: Memory in kilobyte needed to check convoys with 1 to 25 shuttles

## A.5 Comparison

Within this section we compare the results obtained by checking the system with our approach and with checking the system as a whole. Figure 21 shows the differences in consumed time, Figure 22 shows the numbers of generated BDD nodes for both approaches and Figure 23 compares the memory consumption in both cases. Because of scalability reasons we show the convoy results only with 1 up to 10 shuttles.

Without composition it is necessary to check each possible convoy to show the correctness of the system. This is emphazised by the check of the AddCheck2 constraint which is evaluated to true with 1 to 20 shuttles and false when being check for 25 shuttles. But as can be seen within the figures the time and the memory needed to check a system grows very fast. And in our case a check of a convoy with 30 or more shuttles failed. This is the reason why a compositional approach where only relative small parts of a system have to be checked to show the correctness of the whole system is needed.

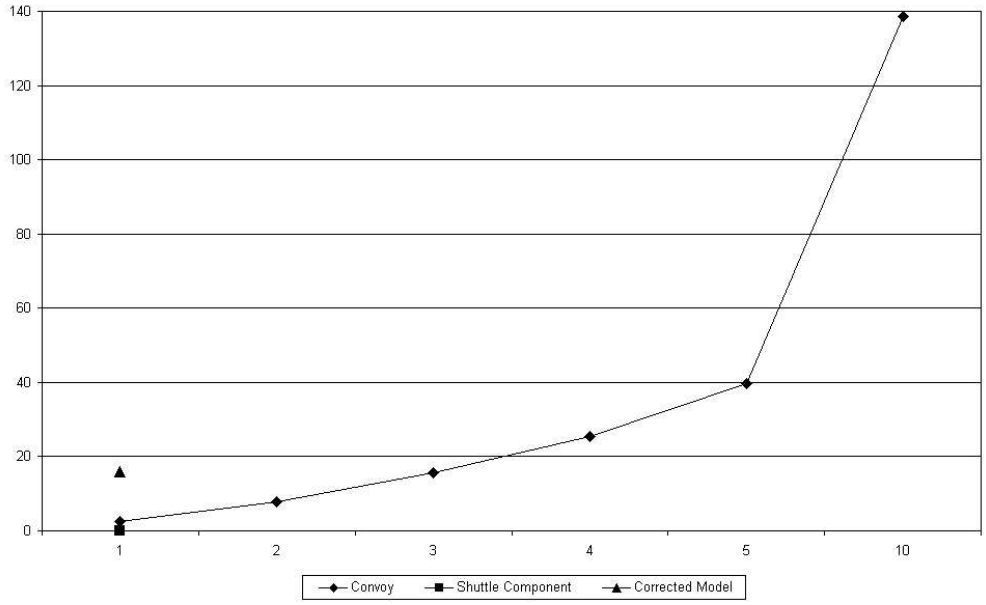


Figure 21: Time needed to check convoys with 1 to 10 shuttles and to check one shuttle component and both versions of the distance coordination pattern as done within our approach.

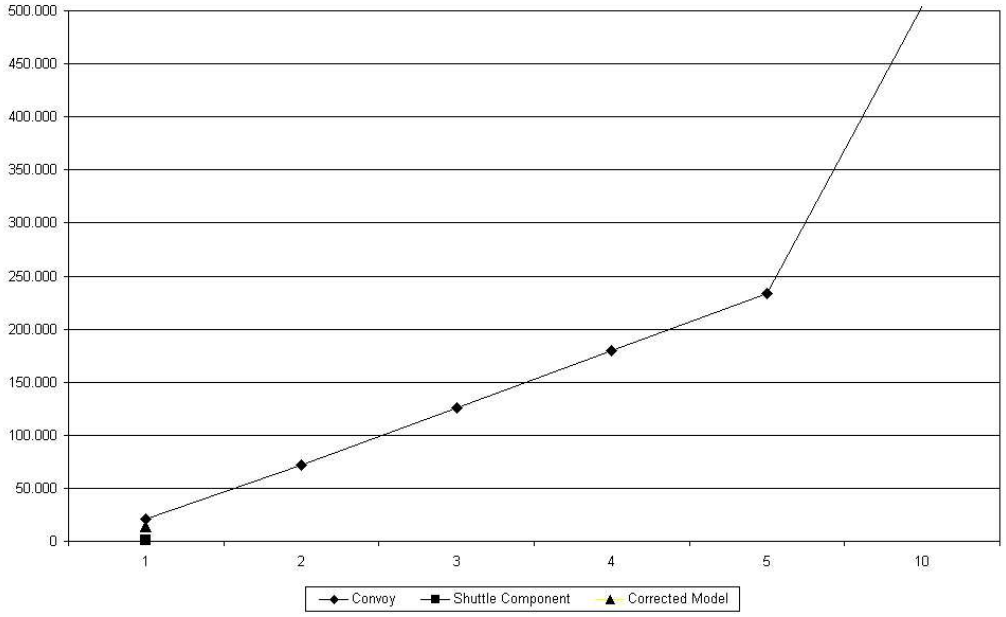


Figure 22: Number of BDD nodes generated when checking convoys with 1 to 10 shuttles and when checking one shuttle component and both versions of the distance coordination pattern as done within our approach.

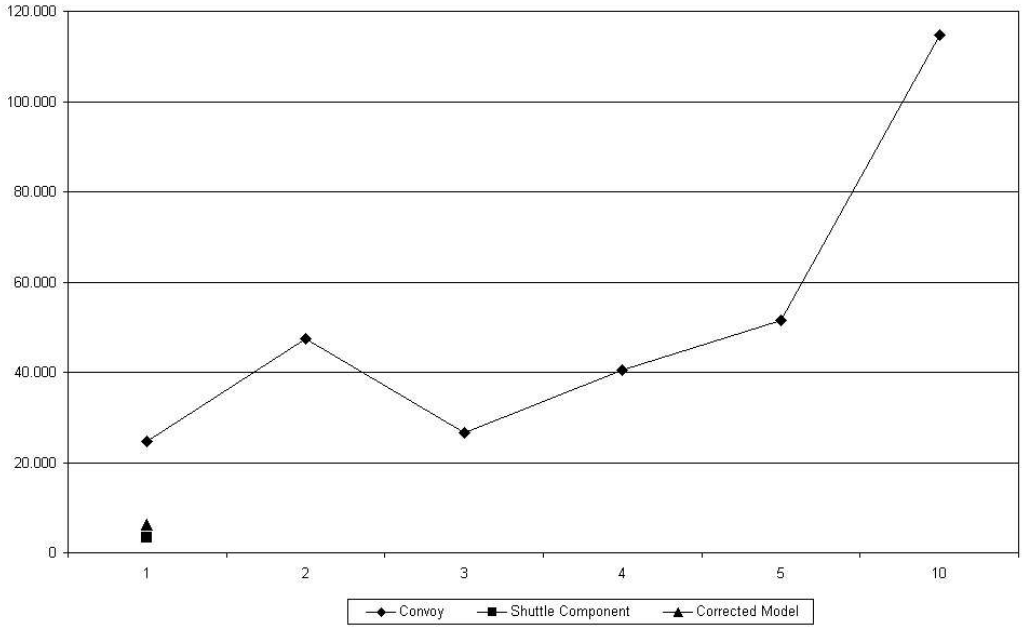


Figure 23: Memory needed to check convoys with 1 to 10 shuttles and to check one shuttle component and both versions of the distance coordination pattern as done within our approach.