Syspect – Modelling, Specifying, and Verifying Real-Time Systems with Rich Data

Johannes Faber, Sven Linker, Ernst-Rüdiger Olderog, and Jan-David Quesel
(Department of Computing Science, University of Oldenburg, Germany)

Abstract   We introduce the graphical tool Syspect for modelling, specifying, and automatically verifying reactive systems with continuous real-time constraints and complex, possibly infinite data. For modelling these systems, a UML profile comprising component diagrams, protocol state machines, and class diagrams is used; for specifying the formal semantics of these models, the combination CSP-OZ-DC of CSP (Communicating Sequential Processes), OZ (Object-Z) and DC (Duration Calculus) is employed; for verifying properties of these specifications, translators are provided to the input formats of the model checkers ARMC (Abstraction Refinement Model Checker) and SLAB (Slicing Abstraction model checker) as well as the tool H-PiLoT (Hierarchical Proving by Instantiation in Local Theory extensions). The application of the tool is illustrated by a selection of examples that have been successfully analysed with Syspect.

Key words: real-time systems; modelling; UML; formal specification; CSP; Object-Z; Duration Calculus; model checking; abstraction-refinement; ARMC; SLAB; H-PiLoT


1 Introduction

Safety critical systems—for instance, traffic assistance systems that should guarantee collision freedom of cars or trains—necessitate the use of formal models of the overall system and of formal verification for establishing relevant safety properties.

These models must be able to represent various aspects of the systems such as state spaces and their transformation, communication between system components, and real-time constraints. To cope with such models in a manageable way, combined specification techniques have been proposed, integrating well researched specification techniques for individual system aspects. One challenge is to make such specification techniques accessible to software engineers. Another (grand) challenge is to develop methods for the automatic verification and analysis of such combined specifications modelling complex real-life systems.

This work was partially supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center “Automatic Verification and Analysis of Complex Systems” (SFB/TR 14 AVACS) and as part of the graduate school “TrustSoft” (GRK 1076/1)
Corresponding author: Johannes Faber, Email: johannes.faber@informatik.uni-oldenburg.de
Received 2010-10-01; Accepted 2011-01-03; Final revised version 2011-01-17.
To meet the first challenge, diagrammatic notations are often used, in particular the ones collected in the Unified Modeling Language (UML). For a formal analysis, such notations need to be backed up by a formal semantics. It is well-known that capturing the semantics of diagrammatic notations can be very difficult. Thus it is not surprising that for a complex language like UML various semantics (of suitable subsets) have been developed. For example, Broy bases a semantics of message sequence charts on streams, the relational calculus of object and component systems (rCOS) bases its semantics on the Unifying Theories of Programming (UTP), and Knapp et al. base the semantics of timed state machines and collaborations for their tool HUGO/RT on timed automata.

We employ a UML profile designed for object-oriented, reactive systems that has been extended to cover real-time aspects. UML profiles classify elements of UML into stereotypes and extend the diagrammatic elements with tags for providing additional information. These profiles are tailored thus for specific system types or application areas. Our profile uses the notations of UML 2.0 and comprises class diagrams, protocol state machines, and component diagrams, annotated by tags. The profile has a formal semantics by translation into the specification language CSP-OZ-DC (abbreviated COD), which combines elements from Communicating Sequential Processes (CSP), Object-Z (OZ) for data, and Duration Calculus (DC) for real-time. To this end, suitable tags of the diagrams in the UML profile represent the contents of classes in the form of OZ and DC expressions.

To address the second challenge, we pursue an automata-theoretic approach where both the system (model) and the property are translated into automata so that model checking is reduced to checking emptiness of languages or the reachability of states. The key for applying this approach to CSP-OZ-DC is an operational semantics of this language based on Phase Event Automata (PEA), which extend timed automata such that the parallel composition synchronises on both phases (state formulae) and events. We consider the problem whether a model in the UML profile or the corresponding specification in CSP-OZ-DC satisfies a real-time property expressed by a DC formula. In general, this problem is undecidable because CSP-OZ-DC specifies infinite-state systems. Thus, we aim at automatic verification methods that are applicable in many interesting cases despite the infinite state space.

According to the automata-theoretic approach both the specification and the real-time property are translated to PEA running in parallel. More specifically, the property $\varphi$ is translated to a test automaton $\text{PEA}_{\text{test}}(\varphi)$, which has a distinguished bad state such that specification $S$ satisfies the formula $\varphi$ if and only if the bad state is not reachable in $\text{PEA}_{\text{test}}(\varphi)$ when running in parallel to the PEA of $S$. To check for reachability, PEA are translated further into Transition Constraint Systems (TCS), which serve as input language of several verification engines that we use: ARMC (Abstraction Refinement Model Checker), SLAB (Slicing Abstraction model checker). Moreover, the TCS representation can be used for invariance or bounded model checking of safety properties with H-PILoT (Hierarchical Proving by Instantiation in Local Theory extensions).

Syspect (System Specification Tool) provides graphic and textual editors for modelling real-time systems with rich data in the UML profile mentioned above. Syspect contains translators to CSP-OZ-DC and furthermore to PEA and TCS, thus pro-
Table 1. Overview of the levels and languages covered by Syspect. The arrows correspond to automatic translation functions provided by Syspect.

<table>
<thead>
<tr>
<th>level</th>
<th>language</th>
<th>purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>modelling</td>
<td>UML profile</td>
<td>model $M$ of a real-time system $R$ with rich data</td>
</tr>
<tr>
<td>specification</td>
<td>CSP-OZ-DC</td>
<td>specification $S$ of $R$ as the formal semantics of $M$</td>
</tr>
<tr>
<td>verification</td>
<td>PEA</td>
<td>operational semantics $O$ of $S$</td>
</tr>
<tr>
<td></td>
<td>TCS</td>
<td>representation of $O$ as input for verification engines</td>
</tr>
</tbody>
</table>

Figure 1. Overview of the levels and languages covered by Syspect. The arrows correspond to automatic translation functions provided by Syspect.

The contributions of this paper are as follows:

- We give an overview of the System Specification Tool Syspect, which is designed for modelling, specification, and verification of real-time systems with rich data.
- To this end, we discuss the language levels involved in Syspect and present the methods and data flow of an automatic verification with Syspect.
- We illustrate these aspects by using excerpts from the case study ETCS (European Train Control System) and briefly describe further case studies that have been treated with Syspect.

Structure of the paper. Using a motivational example, Section 2 introduces the UML profile for formally modelling real-time systems within Syspect. Section 3 explains the underlying CSP-OZ-DC semantics of the UML profile and Section 4 elucidates verification approaches applied in Syspect. Some case studies realised with Syspect are summarised in Section 5 and Section 6 concludes with references to related work.

2 Modelling with Syspect

2.1 Motivational example

To describe the UML profile and the underlying formal model in CSP-OZ-DC, we will use a part of the European Train Control System (ETCS)\cite{10} as our running example\cite{12}. In the case study scenario, an arbitrary number of trains drive on a track. A radio block controller (RBC) supervises all trains in a defined area and grants movement authority ($ma$) to the trains in that area. Every train may drive safely on its track up to its assigned $ma$. If a train is approaching the end of its $ma$ it requests an extension of this authority from the RBC. Taking the distances to and speeds of other trains in account, the RBC calculates the new $ma$ and sends it to the
train. Hence, correct and safe behaviour of the RBC is crucial for the overall safety of the system, in particular for ensuring collision freedom of trains. In the following subsections, we outline the formalisation of this example.

2.2 Modelling the example with the UML profile

For modelling the RBC and the trains, we employ the UML profile mentioned in Section 1[28]. It uses three types of diagrams. Class diagrams (cf. Fig. 2) define the static structure of the system, i.e., which types of objects are capable of communicating with each other. The dynamic behaviour of classes is given in terms of protocol state machines (cf. Fig. 3), in the following just called state machines. Component diagrams (cf. Fig. 4) specify the number of instances of each class and the communication structure between these instances. Stereotypes are used to specialise UML classes to capsules, data classes, and interfaces. In the diagrams, these classes are marked with icons containing C, D, and I, respectively. Thus in Fig. 2, the classes Train, RBC-Com and RBC-Controller are capsules, Environment and MAInterface are interfaces, and SegmentData and TrainData are data classes.

Capsules describe entities with a state space, operations on the state space, a control structure on the operations, and timing constraints. In terms of UML classes, capsules contain attributes and methods, and may be associated with a state machine to define the behaviour of the capsule. Interfaces are used to model the operations used and provided by each capsule for communication and synchronisation. They define which methods are publicly available and their parameters, but not their actual implementations. Data classes describe arbitrarily complex data types containing also operations, but in contrast to capsules, they may not possess a control structure and timing constraints and they are not able to communicate via interfaces.

Capsules are allowed to realise (−↠) and depend (↠) on interface classes (see Fig. 2), thereby implementing all method definitions of the respective interfaces. Further connections between classes are compositions (−→), to express that a class is a part of another.
State machines define the control structure of capsules. They consist of states connected by transitions, which are labelled with operation names. If the state machine is in some state $s$, only operations referred to on outgoing transitions of $s$ may be executed. At the instantiation of a capsule, the state machine is in the initial state, denoted by a filled, black circle. If the state machine reaches a final state, denoted by a hollow circle containing a filled circle, the state machine halts accepting the input. Furthermore, a state may contain several regions separated by dashed lines, which are concurrently executed. The state machine of the train capsule is shown in Fig. 3. Its main part is the state containing three regions, one for the communication with the RBC, one for the control of the trains speed, and one for updating its data structures.

![Figure 3. State machine of Train Capsule](image)

The difference between method definitions and implemented methods is reflected in the profile by different sets of tags. Method definitions possess the tags `in` and `out` defining input and output parameters, while the tag `simple` contains parameters to identify specific instances of capsules. Method implementations have three additional tags. The tags `enable` and `effect` contain OZ-formulae defining the preconditions for the activation of a method and the result of the execution of the method, respectively. The primed (unprimed) occurrences of variables refer to the attributes value after (before) the methods execution. The `changes` tag defines which variables may be affected by the execution of the method. In the example, the `getPos()` method of the capsule `Train` has the precondition $\text{curPos} + \text{curSpd} < \text{ma} \lor \text{nextAllowedSpd} \leq 0$ and the effect $\text{curPos}' = \text{curPos} + \text{curSpd}$. That is, this method is only executable, if the next position of the train is within the given $\text{ma}$ and in that case updates the position of the train by incrementing it by the current speed times the sampling rate which is assumed to be one for simplicity here.

Furthermore, the capsule stereotype defines three types of constraining tags. First, are the `init` and `invariant` tags, where the former defines the state at the instantiation of the class, while the latter defines which constraints have to hold during the existence of an instance. Both tags contain Object-Z formulae. For example, the formula $0 < \text{allowedSpd} \leq \text{gmax} \leq 30$ is a part of the invariant tag of the capsule `Train`. It expresses that the train may move only at a speed $\text{allowedSpd}$ greater than zero.
and at most $g_{max}$, which is at most 30. The third tag is called $dc\ counterexample$ and constrains the allowed behaviour of the capsule with respect to real-time properties, by prohibiting certain traces of behaviour and timing. An overview of tags and their associated diagrammatic elements can be found in Table 1.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Element</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>$in, out, simple$</td>
<td>method def./impl.</td>
<td>parameter definition</td>
</tr>
<tr>
<td>$changes, enable, effect$</td>
<td>method impl.</td>
<td>state space transformations</td>
</tr>
<tr>
<td>$init, invariant$</td>
<td>capsule</td>
<td>initial attribute values</td>
</tr>
<tr>
<td>$dc\ counterexample$</td>
<td>capsule</td>
<td>forbidden timing behaviour</td>
</tr>
</tbody>
</table>

Finally, the component diagram in Fig. 4 shows the objects in the system. The capsule System is a special class representing the system itself, which is needed in case a component diagram is present. The unique System component defines which classes are instantiated with certain multiplicities. Hence, all instances of other capsules are contained in it. The communicating part of the RBC (RBC-Com) and the trains are connected by the MAInterface, while the communication channels with the environment are delegated to the system.

![Component diagram of the system](image)

2.3 Syspect

The tool Syspect (cf. Fig. 5) is designed for modelling real-time systems using the UML profile described in Section 2.2. Modelling typically proceeds as follows. One starts to model the main entities of the system as capsules in some class diagram. The next step is to describe the communication channels by adding interfaces and connecting capsules and interfaces appropriately. Afterwards, the focus switches to the internal behaviour of the capsules themselves and method behaviour is specified using Object-Z schemata. Intertwined with this step is the task of adding state machines to describe the control flow of the system components. Having specified the abstract system behaviour the user can now form concrete instances of the system by adding component diagrams and specify unwanted behaviour using DC formulae.

For these tasks, Syspect provides several editors, a graphical one for every type of diagram (class diagrams, state machines, and component diagrams) and textual
editors for the definition of OZ schemata and types as well as DC formulae to fill the tags described in Sect. 2.2. To input these formulae, Syspect features a formula editor that provides a clickable list of Z-specific symbols as well as on-line text-replacement that substitutes \LaTeX-input on-the-fly with corresponding Unicode characters representing the special symbols. This both supports intuitive understanding of the typed formulae as well as convenient inputting of those.

In general, the diagrams provide specific views on a common model of the specification. For example, this enables the user to create two different class diagrams, the one in Fig. 2 and a second one describing the aggregation of the three capsules to the system class, where all details of the capsules (attributes, methods) and their structural connections are omitted. For the sake of clarity, also the inner contents of components and states in a state machine may be removed from the view and specified in different diagrams.

As all these diagrams build a common model that has to be accessible by the user, Syspect includes two different ways to browse the specification. On the one hand, the user can edit the diagrammatic elements and create new views on existing model elements. On the other hand, Syspect allows for the direct manipulation of model elements, i.e., adding new methods to a capsule without having to search it in some diagram first.

In Fig. 5, a screenshot of Syspect is provided. In the screenshot the class diagram editor and below the property view, where the tags can be filled, are shown. On the left one can see the diagram explorer which shows all diagrams and diagram elements. Another tree view, the model explorer, is hidden behind it. This one provides a view on all elements of the background model.
2.4 Export functions

An important task when dealing with system specifications is to process them further. To this end, Syspect provides two types of export functions. On the one hand, there are exports for documentation purposes. These are the different image exports for the diagrams and a LaTeX export for the underlying COD semantics. These exports help the user in producing documents and getting an overview of the system semantics. For example, the image export as well as the LaTeX export were used to generate most figures in this paper. On the other hand, Syspect features further exports of its semantics into different representations.

2.5 Tool structure

Syspect is implemented in Java as a plug-in structure upon the Eclipse framework. This for one enables builds for many different platforms as well as profiting from usability features already provided as stubs by the framework. For example, Syspect makes use of the drag and drop features for diagram elements and the possibility to rearrange the Syspect window to a user-preferred setting.

The plug-in architecture provides an easy way to add further exports to the tool and link into the verification process, and it enables the user to choose which flavor of the whole Syspect environment she likes by disabling unused parts. There is for instance an optional plug-in that enables the user to visualise and edit the PEA level of the specification. This feature is especially useful when one wants to test out optimisations to the constructed semantics before applying a model checker.

3 Formal Specification with CSP-OZ-DC

For verification purposes, a semantics in terms of a formal language is needed. In this section, we show an exemplary formalisation of the example in the combined specification language CSP-OZ-DC, which integrates CSP\cite{19} for the description of control structure, Object-Z\cite{7} to specify data changes over infinite data types and parameters, and Duration Calculus\cite{42} for constraining the system’s realtime behaviour. As has been shown in Ref. \cite{28}, systems modelled with the UML profile described in the previous section can be translated into COD specifications. The system we show results from translating the ETCS model into COD with the help of Syspect. For a description of the translation, see Section 4.

Figure 6 shows a part of the RBC specification and Fig. 7 the system process. The full specification is given in Ref. \cite{12}. A COD class is divided into several parts as shown in Fig. 6: the interface, the CSP part, the OZ part consisting of data classes, state, init schema, and operation schemas, and finally, (not present in Fig. 6) the DC part. The interface defines the parameters of the methods defined in the class as well as which methods are exposed to the outside of the class (method) and which methods are used only internally (local_chan). The CSP part defines the dynamic behaviour of the class, i.e., in which order methods may be executed. Furthermore, classes may be instantiated by referencing their name in the CSP part. For example, in the system class (cf. Fig. 7), we find a structure reflecting the component diagram in Fig. 4. Four classes are instantiated. The two Train classes and the RBC-Com

\[\text{Available at http://www.eclipse.org}\]
communicate over three channels. To prevent other instances from interfering with these communications, the channels are renamed, i.e., not accessible by processes outside the scope of the renaming operator [old/new].

Data classes serve as OZ-type definitions for complex data types. In the unnamed schema, both the attributes of the class and the invariant holding throughout the existence of instances of the class are defined, while the Init schema gives the possible initial values of the attributes (cf. “State and Init schema” in Fig. 6). The operation schemas define the pre- and postconditions of the methods, which are given by the enable and effect schemata, respectively.
Figure 7. COD Specification of the system process

Finally, the DC part contains DC counterexample formulae restricting the execution of methods with respect to real-time properties. E.g., the formula

$$\neg (\text{true} \uparrow \uparrow \text{getPos} \uparrow (\ell < 2) \uparrow \uparrow \text{getPos} \uparrow \text{true})$$

is a part of the train specification, ensuring that at least 2 time units pass between two getPos events. See Section 4 for a more detailed description of DC formulae.

4 Verification with Syspect

A major goal of Syspect is to provide access to novel formal verification approaches through suitable plug-ins. Since every model in the UML profile can be transformed into a COD specification, which in turn has a semantics in terms of PEA and Transition Constraint Systems (TCS)\(^{[21]}\), the main approach in Syspect is to employ verification tools that use TCS as their input language. Additionally, there is an implementation that uses the CSP-OZ semantics\(^{[13]}\) in terms of CSP by which verification with the FDR model checker for CSP is possible. The remaining section focuses on the TCS verification approach and decompositional verification techniques that are implemented in Syspect to simplify verification tasks.

4.1 Transition constraint systems

This subsection deals with the main approach to verify Syspect specifications via TCS. Figure 8 shows the translation steps and tools involved in the translation to TCS. As explained in Section 2, the UML diagrams of Syspect have—via a UML profile—a direct correspondence to COD elements\(^{[28,38]}\). The translation from UML to COD proceeds as follows:

- Capsules are translated into COD classes. Attributes and methods of capsules directly become attributes and methods of the class. This is possible because the methods of capsules are already in OZ syntax with a dedicated changes list of symbols that can be changed in a method and with dedicated input and output variables.
- Data classes are translated into OZ classes—a class without control structure nor timing part.
- State machines in Syspect always belong to a capsule and can be translated into equivalent CSP processes.
• DC formulae also belong to capsules and can directly be conveyed to the corresponding COD class.

These translation steps are automatically executed in Syspect.

![Translation to PEA](image)

**Translation to PEA.** Since the COD model is still a declarative model, it is translated in a second step into an operational model, which is more suited for automated verification by model checking. To this end, Hoenicke\cite{21} and Meyer et al.\cite{26} give a translation from COD into an extended timed automata model, so-called Phase Event Automata (PEA), that synchronise on both events and data. That is, a variable in a parallel component can only be changed if all parallel components agree. We do not go into the details of this translation here but only give its main features. We refer the interested reader to Ref. [21] and Ref. [26] for detailed explanation and to Ref. [14] for an overview.

The translation is compositional in the sense that every part of the COD specification is translated on its own:

\[
PEA(CSP-\text{OZ-DC}) = PEA(CSP) \parallel PEA(\text{OZ}) \parallel PEA(\text{DC})
\]

if \(CSP-\text{OZ-DC}\) is a COD specification with the corresponding parts \(CSP, \text{OZ}, \text{DC}\). The DC part generally consists of several DC formulae \(DC_1, \ldots, DC_n\) that are each
translated into a single automaton, i.e.,

$$PEA(DC) = PEA(DC_1) \parallel \ldots \parallel PEA(DC_n)$$

The translations of the CSP and the OZ part are simple. The CSP process is translated into its operational semantics\(^\text{[34]}\) (thus, it has to be finite to be verifiable by model checking) in terms of PEA without data constraints and time. The OZ part is translated into an automaton with two locations, one initial location, having the init schema of the COD specification as invariant, and a second location with a loop transition for every operation of the COD class. The state invariant of the COD class becomes invariant of both of the locations. So, the constraints of the operation schema are directly passed to the transitions and invariants of the PEA.

The difficult part is the translation of the DC formulae. Since the full DC is undecidable\(^\text{[41]}\), it is only possible to translate a dialect into PEA. Hoenicke\(^\text{[21]}\) identified counterexample traces as a translatable subset. Counterexample traces are negated DC traces of the shape

$$\neg (phase_0 \triangleright a \triangleright \ldots \triangleright phase_n),$$

where $$phase_i = [\varphi] \land \boxdot b \land \ell < T$$ with $$\sim \in \{\leq, <, >, \geq\}$$. The chop operator $$\triangleright$$ divides traces into time intervals; a phase $$[\varphi] \land \boxdot b \land \ell < T$$ means that the constraint $$\varphi$$ is valid during the interval with a length ($$\ell$$ always refers to the length of the current interval) smaller than a rational time constant $$T$$, where no event $$b$$ occurs. The DC formula

$$\neg (true \triangleright \downarrow getPos \triangleright (\ell < 2) \triangleright \downarrow getPos \triangleright true) \tag{1}$$

taken from the train capsule specification states that it is not possible that there is an interval with length smaller than 2 between two $$getPos$$ events or, formulated positively, at least 2 time units pass between two occurrences of $$getPos$$ events.

When translating such a formula into an automaton the resulting automaton has to decide whether a run matches all of the consecutive phases of a trace formula. The problem arises that the consecutive phases are not mutually exclusive in the sense that at a specific point in time several phases are potentially active. For instance, given formula (1), when the automaton detects the first $$getPos$$ event then the second trace ($$\ell < 2$$) can potentially be active or the first $$true$$ phase can be active. Thus the automaton has to reflect all possible combinations of phases. To this end, the general translation needs a power-set construction similar to the construction of a deterministic automaton from a non-deterministic one.

**Compositional Reasoning.** An important consequence of the compositional semantics of COD in terms of PEA is that it permits compositional reasoning in the following sense: whenever a subset of PEA in a parallel composition satisfies a safety property (given by a DC formula $$\varphi$$) then also the full parallel composition does so. This allows for a cone-of-influence verification technique. For example, it suffices to verify a safety property for a single component to conclude safety of the entire system. More formally, according to Ref. [21] a system $$PEA_1 \parallel PEA_2$$ has the property that

$$PEA_1 \models \varphi \ \text{implies} \ \ PEA_1 \parallel PEA_2 \models \varphi$$
for all possible $\text{PEA}_2$. Thus, for a COD system $S$ we can verify every property $\varphi$ that only depends on the timing of the system by verifying $\text{PEA}(\text{DC}) \models \varphi$ to conclude the correctness of $\varphi$ for the entire specification, i.e., $\text{PEA}(S) \models \varphi$. Analogously, we can verify properties only depending on the CSP or the OZ part.

**Translation into TCS.** In a further step, the PEA are translated into Transition Constraint Systems (TCS), simple transition structures that are supported by the verification tools ARMC$^{[32]}$ and SLAB$^{[2,9]}$. A TCS is a tuple $T = (\text{Var}, \text{Init}, \text{Trans})$, where $\text{Var}$ is a set of unprimed variables, $\text{Init}$ is an initial constraint describing all initial states of the system, and $\text{Trans}$ is a transition constraint describing state changes, where—as in Z and Object-Z—unprimed variables refer to the state before the change and primed variables to the state after the change. At the level of TCS, the clocks of PEA are represented as real-valued data variables, following the “old-fashioned recipe” advocated by L. Lamport.

The translation from a PEA to a TCS is described in Refs.$^{[17, 21]}$ and it is relatively straightforward in that the only constructs in PEA that deserve a special consideration here are the PEA locations and the clock constructs. The remaining constructs like state changes during PEA transitions can directly be treated as TCS transitions because PEA (and likewise COD) use also a primed and unprimed constraint notation to describe state changes. PEA locations are encoded in a TCS using a program counter variable and clocks of PEA are encoded by real-valued variables. Since clock constraints in PEA are always convex, the progress of time can be modelled using a further real-valued variable that is added to the clocks in each transition step. For details see Refs. $^{[17, 21, 14]}$.

The drawback of this procedure is the state space explosion because a system that consists of parallel components has to be translated into a single TCS (Fig. 8) since at present the model checkers ARMC and SLAB can process only a single TCS.

**Implementation.** As depicted in Fig. 8, the translation to a COD specification is executed within Syspect. Computing the parallel product of all PEA and translating it into a TCS is done by the PEA toolkit$^{[26]}$, which is part of Syspect but also available as a separate library for the handling of PEA. The translation of DC formulae to PEA is also implemented in the PEA toolkit$^{[2]}$.

### 4.2 Verification of Syspect specifications

**Verification with SLAB/ARMC.** The model checker ARMC$^{(3)[32]}$, developed at the Max-Planck-Institut für Informatik in Saarbrücken, and SLAB$^{(4)[2,9]}$, developed at the Saarland University, both take a single TCS as input. They verify safety properties of infinite-state systems by checking reachability of bad states and return counterexample traces if these bad states are reachable.

In Refs.$^{[25, 26, 27]}$, these bad states are specified using DC formulae that are also translated into TCS. To this end, the DC counterexample trace formulae are extended to DC test formulae that can be used to specify bad behaviour that shall be avoided by a correct implementation. The PEA toolkit also implements translation of test formulae into PEA such that ARMC and SLAB can be used to verify Syspect

---

2) Available at [http://csd.informatik.uni-oldenburg.de/projects/epea.html](http://csd.informatik.uni-oldenburg.de/projects/epea.html)
3) Available at [http://www7.in.tum.de/~rybal/armc/](http://www7.in.tum.de/~rybal/armc/)
4) Available at [http://react.cs.uni-sb.de/tools/slab.html](http://react.cs.uni-sb.de/tools/slab.html)
specifications against DC test formulae by translating the specification and the test formulae into PEA and passing their parallel composition to the model checker.

Using the TCS verification approach, Syspect enables parametric verification of real-time systems in the data and the time dimension\cite{14}. That is, it is not necessary to give specific values for all system parameters; it suffices to constrain system parameters to guarantee safety.

As explained in Section 4.1, all constraints occurring in Syspect specifications are directly shifted down to the TCS level (modulo syntactical modifications). Since Syspect and COD both allow arbitrary Z expressions in operation schemes to describe state changes, not every valid Syspect specification can be verified with ARMC or SLAB, which both can handle only linear arithmetic constraints over real-valued variables without quantifiers. The design decision for Syspect was not to restrict the Syspect language to a specific COD subset that can be verified but to allow a very expressive language at the level of Syspect, which is also supported by COD and PEA. The advantage of this is that Syspect can be used to design specifications even though they cannot be verified. Moreover, Syspect can easily be extended by further verification plug-ins without the need to adapt the Syspect language itself—only the way how the specification is treated needs to be implemented. An example for a verification plug-in that allows for more complex input constraints than SLAB or ARMC is given with the H-PILoT plug-in, described below.

**Verification Feedback.** Syspect supports this verification approach by allowing users to specify correctness properties of a model by DC test formulae. Using convenient input dialogues, a user may export a TCS for a given test formula that can directly be verified with SLAB or ARMC. In addition, ARMC can be called from within Syspect. In this case, the user also gets the feedback from the model checker within Syspect. If ARMC returns a counterexample trace it is of course related to the transition constraints but these usually do not contain information about the corresponding Syspect elements—this information gets lost during the translation and construction of the parallel composition. For this reason, there is a Syspect plug-in\cite{20} that maps the counterexample trace provided by ARMC back to the corresponding high-level elements of the Syspect model (cf. Fig. 8): a counterexample view is displayed that allows step-wise following the trace to the error. This way, a user can reproduce the origin of a system error and correct the model accordingly.

**Verification with H-PILoT.** To support more complex data structures, Syspect additionally interfaces with the verification tool H-PILoT\cite{22} that has been developed at the Max-Planck-Institut für Informatik in Saarbrücken. H-PILoT uses hierarchical reasoning in chains of local theory extensions to reduce the complexity of verification tasks. To this end, the satisfiability of constraints over specific theory extensions that are identified to be local are reduced to the satisfiability of constraints in a base theory for that a dedicated prover exists. Standard SMT solvers can then be used to check the satisfiability of the formulae of the base theory. With this approach, the invariant checking problem for local theory extensions becomes decidable. In this way, COD specifications with properties over rich data types like arrays\cite{14} or pointer data structures\cite{12} can be verified.

\footnote{Available at http://www.mpi-inf.mpg.de/~ihlemann/software/index.html}
In Fig. 6, a linked list is modelled with a function \textit{next} from \texttt{Train} to \texttt{Train}. Changes or invariants over this list are modelled with quantified expressions like

\[ \forall t : \texttt{Train} \bullet \text{next}(\text{prev}(t)) = t. \]

The safety property that is to be verified is that the RBC controller never assigns one segment to two different trains, which can be expressed by the invariant property

\[ \forall t_1, t_2 : \texttt{Train}. \ t_1 \neq t_2 \rightarrow \text{sid}(\text{segm}(t_1)) \neq \text{sid}(\text{segm}(t_2)). \]

With the hierarchical reasoning approach, such invariant checking problems are reduced to a decidable fragment. The advantage of having a decidable problem is that the solver returns a result for correct and incorrect verification tasks. In the case that a verification task fails H-PILoT returns—with the help of the underlying solver—a model that violates the desired property. This helps the user to correct the model.

Syspect also allows a user to define safety properties as an \textit{invariant} that needs to hold for all possible transition steps of a Syspect specification. Syspect generates a TCS from the Syspect model and the desired invariant property. The PEA toolkit outputs this TCS as a set of proof tasks in H-PILoT syntax that can be handed over to H-PILoT. Since H-PILoT demands some meta information about the data types of the Syspect specification, like, e.g., the corresponding local theory extension for the specification’s function symbols or the symbols that should be treated as pointer functions, Syspect provides a dialogue to automatically hand over these information to the prover for each proof task.

Like the ARMC/SLAB verification approach, H-PILoT verification is parametric for data and time parameters. Moreover, as H-PILoT allows for verifying complex pointer structures or arrays, it is also possible to verify a parametric number of components, e.g., an arbitrary number of trains in the RBC example from Sect. 2.1.

4.3 Decompositional verification in Syspect

Syspect is designed to cope with systems with concurrent components, real-time, and (possibly) infinite data types like lists. Due to this inherent complexity of the specifications written in Syspect, direct verification is often not possible because of the well-known state space explosion problem. Thus, further decomposition techniques are important to enable verification of large and realistic models. In this section, we elucidate on the decomposition approaches that are implemented in Syspect.

\textbf{Slicing CSP-OZ-DC Specifications.} The technique of \textit{slicing} program specifications is well-known for reducing the state space of systems. The idea of slicing is to syntactically decompose a specification with respect to a safety property into relevant and irrelevant parts. If this decomposition is sound, only the relevant parts have to be verified with model checkers. Brückner has lifted slicing to the level of COD specifications\cite{5}.

For a given a COD specification \( S \), first the \textit{control flow graph} (CFG) of \( S \) is computed, where the nodes represent the operational parts of the specification, i.e., the CSP operators and OZ methods, and where the edges define the possible control flows. Based on this CFG, the dependencies between the possible method executions are examined and the \textit{program dependence graph} (PDG) is created. The edges of the PDG represent different types of dependencies, e.g., \textit{direct data dependencies}. A
direct data dependency exists between two methods \( m_1 \) and \( m_2 \) if the execution of \( m_1 \) changes the values of variables which are referenced in the precondition of \( m_2 \).

The PDG is subsequently analysed with respect to a slicing criterion given as a DC test formula as described in Section 4.2. The analysis creates a backward slice of the specification, i.e., a new specification which consists only of the parts influencing the property defined by the slicing criterion. The state space of the backward slice is in many cases significantly smaller than that of the original specification, as the experiments by Brückner have shown.

When verifying the train component from the running example in Sect. 3, the slicing criterion \((true \land [\mathit{carPos} > \mathit{ma}] \land true)\) is used, defining the unwanted situation that the train exceeds its movement authority. Computing the backward slice of the example results in a smaller specification by removing two variables (the identifier of the train and an additional auxiliary variable) not involved in the train movement.

Syspect contains a slicing plug-in implementing this approach. The slice created by the plug-in can be either exported like a full specification, or verified with the supported model-checkers. In the latter case, the slice has to be checked against the slicing criterion, to preserve correctness of verification results.

Verification Architectures. Moreover, Syspect supports compositional verification by applying the Verification Architecture approach\[11\]. A Verification Architecture (VA) is an abstract behavioural protocol that splits the system runs into several phases (in the train example this could be, e.g., a braking phase and a running phase) with local real-time properties that hold during these phases. After showing the correctness of a desired global property for this protocol, the global property is also guaranteed by all instances of the protocol for that the local real-time properties are satisfied. That is, given a correct VA we can verify global properties of concrete models by combining local analyses: for a concrete model it needs to be checked that the protocol structure corresponds to the structure of the VA—a step that can be done purely syntactically—and it needs to be verified, e.g., by model checking, that the local real-time assumptions are valid for the corresponding components of the concrete specification. For more detailed information on VAs see Ref.\[11\].

VAs can be specified within Syspect\[31\] using an extended state machine editor that allows defining protocol phases and local real-time properties. Given a concrete Syspect specification it can automatically be verified that its structure actually refines a VA. In addition, the verification of the local real-time assumptions is also done automatically by using ARMC or SLAB (Section 4.2).

4.4 Further Verification Plug-Ins.

Furthermore, there are several verification plug-ins for Syspect that have been developed at the University of Paderborn.

A CSP verification plug-in translates a Syspect model without timing constraints to a CSP-OZ\[13\] specification. CSP-OZ has a semantics in terms of CSP\(_M^{[36]}\), a machine readable CSP dialect that can be used as input for the model checker FDR\(_2^{[6]}\) for CSP, which can directly be called from within Syspect. The property that is to be verified needs to be given as a CSP\(_M^{[6]}\) expression.

\[http://www.fsel.com/software.html\]
In Refs. [24, 29], a technique has been introduced to decompose parallel CSP-OZ specifications into two phases using a user-specified cut point. These phases are then verified with a learning-based assume-guarantee rule. This decomposition approach has also been implemented in Syspect and can be used to verify specifications without timing constraints using FDR2.

5 Evaluation

In this section, we report on case studies that have been successfully analysed with Syspect. Most of them have their origin in the railway domain and treat parts of the European Train Control System (ETCS) [10], which is one of the major case studies in the AVACS project. The common setting of these is—as in our introductory example from Section 2.1—that an RBC controller supervises all trains in a defined area and grants movement authorities (ma) to trains. The system is considered safe if the RBC grants these authorities in a sound way, i.e., without overlapping, and the trains never proceed their given ma. This general case study theme occurs in several variants with different characteristics.

Emergency Messages in the ETCS. Reference [5] presents the Syspect model for a case study from Refs. [15, 26] modelling the treatment of emergency messages in the ETCS. The case study has been manually decomposed and verified in Ref. [26]. Using this decomposition, Brückner [5] analysed how the slicing plug-in of Syspect can be used to reduce the model according to the verification tasks of Ref. [26]. He showed that the original model can be reduced by 40–60% for each verification task.

RBC controller for complex track topology. In Ref. [12] the case study from Section 2.1 has been presented. It models an untimed RBC controller that maintains an unbounded linked-list of track segments and an unbounded list of consecutive trains. The trains may also enter and exit the area at any track segment, in contrast to the simpler variant in Ref. [14] where an unbounded array has been used to model a list of trains. Additionally, the Syspect model also contains a timed train controller that behaves according to the certain preconditions postulated in the RBC controller. By showing the correctness of the train controller, we can ensure that the preconditions of the RBC can actually be met by the train controller.

The safety properties to be guaranteed for this combined model are that the RBC never assigns the same track segments to more than one train (cf. Section 4.2) and that the train controllers ensure that the train stays within its movement authority.

Using the compositionality of the COD approach (Section 4.1), the verification of the entire system can be split into several verification subtasks: if one component is proven correct, this property also holds for the complete system. Syspect has been used in Ref. [12] for two verification steps. Firstly, the H-PiLoT export of Syspect has been used to verify the RBC controller with its pointer data structures with H-PiLoT in about two minutes. Additionally, some consistency checks were performed. Secondly, for the timed train controller the ARMC plug-in is used. The resulting TCS consists of over 3300 transitions with 28 real-valued variables and clocks (so it is also an infinite state system). For this reason, the verification with the model checker

---

http://www.avacs.org
ARMC was successfully completed in an order of 26 hours. These verification steps are supported by Syspect by providing appropriate dialogues for specifying the correctness properties and for selecting the components for the verification steps. For detailed verification results, we refer to Ref. [12]. In future, the Verification Architecture plug-in could be used to automatically show that the interaction of train and RBC controller is correct. Currently, this is done manually.

**Train Control Architecture.** Another train control system has been studied in Ref. [11]. An abstract verification architecture for a system where a single train interacts with an RBC controller has been introduced. A deductive approach is applied in Ref. [11] to check that the architecture ensures that every train controller instantiating the architecture stays in its movement authority.

In addition, a concrete model instantiating the verification architecture is presented. To verify that this model is a correct instance of the architecture some local real-time properties needs to be verified for which the ARMC verification plug-in has been successfully used. The refinement relation between the architecture's process structure and the concrete model is established with the VA plug-in from Ref. [31].

**Two phase commit protocol.** A different kind of case study is the (untimed) Two-Phase-Commit-Protocol—a protocol to guarantee consistency of local sites of a distributed database—which is presented in Refs. [24, 29]. It demonstrates the successful application of the cut decomposition approach by verifying the protocol with FDR2 and comparing verification times for the original and the decomposed model. For a detailed discussion, we refer to Ref. [24].

### 6 Summary and Related Work

We presented an overview of the tool Syspect that builds a bridge from models of real-time systems with complex, possibly infinite data expressed in a dedicated UML profile via formal specifications expressed in the combination CSP-OZ-DC to novel verification techniques for infinite-state systems. Current research addresses the enhancement of the verification techniques.

Many tools have been developed that support system development using UML. However, when specification and formal verification of real-time, reactive systems is an issue, there are much fewer approaches available.

In the research group of M. Broy at TU Munich the case tool AutoFOCUS has evolved to a mature tool for modeling and analyzing the structure and behavior of distributed, reactive, and timed computer-based systems [35]. It is based on the theory developed in Ref. [6], and supports system modeling with various types of diagrams and allows for simulation, verification, and code generation. For verification, translations to the model checkers SMV and SPIN are provided. Unlike Syspect, AutoFOCUS is not geared towards verifying real-time systems with complex data.

At UNU-IIST in Macau a method of refinement of components and object systems (rCOS) together with some tool support has been developed [5]. Semantically, rCOS is based on the Unifying Theories of Programming [16]. A strong point of rCOS are refinement rules that for example enable the user a formal analysis of object-oriented design patterns. Verification involves using the FDR model checker, e.g., for checking the consistency of interfaces. In contrast to Syspect, the rCOS approach does not yet
provide tool support for verifying systems with complex data.

The Process Analysis Toolkit (PAT)[37] is a framework for the specification language CSP#, combining CSP processes with a procedural low-level programming language (in C# syntax) and a number of real-time patterns to specify deadlocks, timed interrupts, and time-outs. PAT is designed for verifying LTL-X specifications with fairness and it supports bounded model checking and refinement checking for CSP# programs. Unlike Syspect, PAT does not have a diagram-based input language and, except for the real-time patterns, it does not support infinite-data verification.

The Real-Time Maude Tool[30], based on a rewriting logic approach, also allows for the specification and verification of object-oriented real-time systems. While Real-Time Maude does not employ diagrams for the specification of systems, it is more flexible because it empowers the user to define arbitrary communication models.

**Availability.** Syspect is an open source project distributed under the GNU Public License (GPL) and its sources as well as pre-compiled binaries for Windows, MacOS, and Linux are available at [http://syspect.informatik.uni-oldenburg.de/](http://syspect.informatik.uni-oldenburg.de/). The pre-compiled packages come along with some of the examples from Section 5.

**Acknowledgements**

Syspect has been developed on top of a one-year collaborative student project and several Master theses. It has been further improved in Oldenburg as part of the project “Beyond Timed Automata” of the SFB/TR 14 AVACS and in the research group “Specification and Modelling of Software Systems” of Heike Wehrheim in Paderborn. We are particularly grateful for contributions of the following people: Janna Arnold, Ingo Brückner, Dominik Denker, Christian Günther, Niels Hapke, Jürgen Happe, Klaus Herbold, Ulrich Hobelmann, Patrick Kuballa, Florian Marwede, Björn Metzler, Sebastian Micus, Michael Möller, Matthias Peters, Meik Piepmeyer, Andreas Schäfer, Jannis Stachowiak, Henning Rohlfis, Boris Rosenow, Christian Wenzel, and Daniel Wonisch.

**References**


A Meetings with Manfred Broy

My first encounter of Manfred Broy was in January 1979 at a meeting in Oberwolfach on “Mathematical Methods of Software Construction” organised by H. Langmaack, E. Neuhold and M. Paul. It was my first international meeting in Computer Science, and I saw many established and young scientists which should accompany me throughout my scientific life.

For many years, we had pleasant opportunities of discussing related topics at the annual meetings of IFIP WG 2.2. In the early 1980’s Manfred presented there his elegant theory of stream-processing functions, which solved the Brock-Ackermann anomaly arising in nondeterministic Kahn networks. I always enjoyed talking to Manfred; there was some wonderful light tone in these talks, which often caused us to laugh. We even succeeded writing a joint paper on trace-oriented models of concurrency where we brought together the synchronous and asynchronous view on communicating processes[3].

Among the many meetings over the years, there is one exceptional occasion that connected our otherwise quite different scientific careers in one beautiful event. It was January 1994 in Bonn when we shared one of Leibniz Prices awarded by the German Research Foundation (DFG). Another remarkable event took place in June 2000 in Manfred’s office in the old, commanding building of the Technical University of Munich. In the presence of H. Wössner of Springer-Verlag, Manfred handed over to me the managing editorship of Acta Informatica.

It is difficult to accept how quickly the time of our lives progresses. I wish Manfred all the best for the years to come, and look forward to future meetings.

Ernst-Rüdiger Olderog