A Multi-Paradigm Approach Supporting the Modular Execution of Reconfigurable Hybrid Systems*

Holger Giese,[1] Stefan Henkler[2] and Martin Hirsch[2]

[1]System Analysis and Modeling Group, Hasso Plattner Institute at the University Potsdam,
Prof.-Dr.-Helmert-Str. 2-3, D-14482 Potsdam , Germany
holger.giese@hpi.uni-potsdam.de

[2]Software Engineering Group, University of Paderborn,
Warburger Str. 100, D-33098 Paderborn, Germany
[shenkler—mahirsch]@uni-paderborn.de

Abstract

Advanced mechatronic systems have to integrate existing technologies from mechanical, electrical and software engineering. They must be able to adapt their structure and behavior at runtime by reconfiguration to react flexibly to changes in the environment. Therefore, a tight integration of structural and behavioral models of the different domains is required. This integration results in complex reconfigurable hybrid systems which execution logic cannot be directly addressed with existing standard modeling, simulation, and code generation techniques. We present in this paper how our component-based approach for reconfigurable mechatronic systems, MECHATRONIC UML, efficiently handles the complex interplay of discrete behavior and continuous behavior in a modular manner. In addition, its extension to even more flexible reconfiguration cases is presented.

Keywords Simulation, Code Generation, Hybrid Systems, Reconfigurable Systems

1 Introduction

When developing advanced mechatronic systems we do not only have to combine technologies from mechanical, electrical and software engineering, we also have to develop systems which can react flexibly to changes in the system itself or the environment. Therefore, advanced mechatronic systems have to be able to adapt their structure and behavior at runtime (reconfiguration). Additionally, mechatronic systems usually have real-time requirements and often show hybrid behavior, which requires the integration of the different modeling paradigms in the form of a hybrid system. Due to the increasing complexity, such integration has to cover advanced specification techniques from the involved disciplines as well as their tools.

The design of complex mechatronic systems has become so intricate that it can only be done by means of computer-aided modeling [30]. The models comprise modules and hierarchies that are derived from the physical-topological structure of the system. For subsequent symbolic and numerical processing, the models are transformed into a format appropriate for processing which takes their modular-hierarchical structure into account [61].

Hierarchical block diagrams are the usual method to model technical or reactive systems. They are used in different domains, e.g., in mechanical and electrical engineering, software or systems engineering. This

*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.

common notation has its origin in control engineering. The blocks describe the behavior of the system. Modularization and hierarchical structuring are important aids for the solution of the complexity problem in technical systems. Modularizing the function helps to decompose it into isolated sub-problems which can be addressed by different working teams and different suppliers. In software engineering, component-based specification techniques that emphasize in addition the component type and interface typing rather than classical block diagrams are used to model the required hierarchical decomposition.

In addition, only a single paradigm, such as continuous systems from the control engineering domain, is not sufficient to address advanced mechatronic systems. Instead, multiple paradigms from the involved domains have to be supported. Mosterman and Vangheluwe [51] have identified three orthogonal dimensions for multi-paradigm development approaches, which are i) models of different abstractions, ii) different formalisms, and iii) meta-modeling. For the reconfiguration within advanced mechatronic systems as targeted by this article, in particular, the integration of different formalisms matter.

Advanced Mechatronic System Example A typical example of an advanced mechatronic system is the RailCab[1] research project at the University of Paderborn. In this project, autonomous shuttles are developed which operate individually and make independent and decentralized operational decisions.

The modular railway system combines sophisticated undercarriages with the advantages of new actuation techniques as employed in Transrapid[2] to increase passenger comfort while still enabling high speed transportation. In contrast to Transrapid, the existing railway tracks will be reused [58].

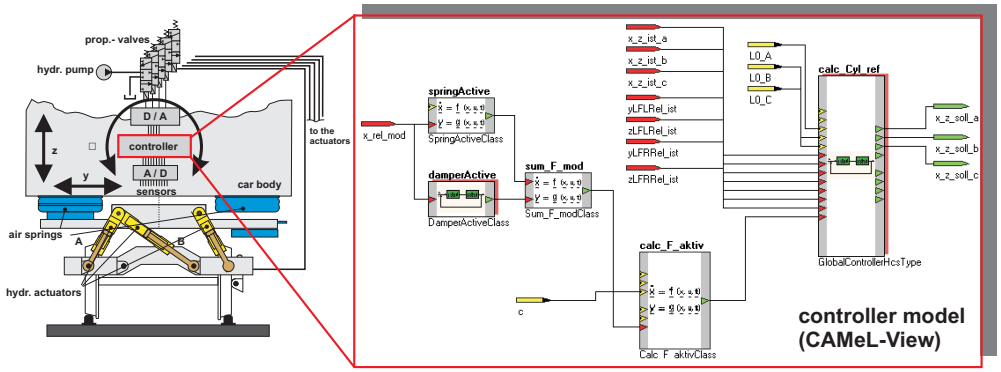


Figure 1: Suspension module with controller

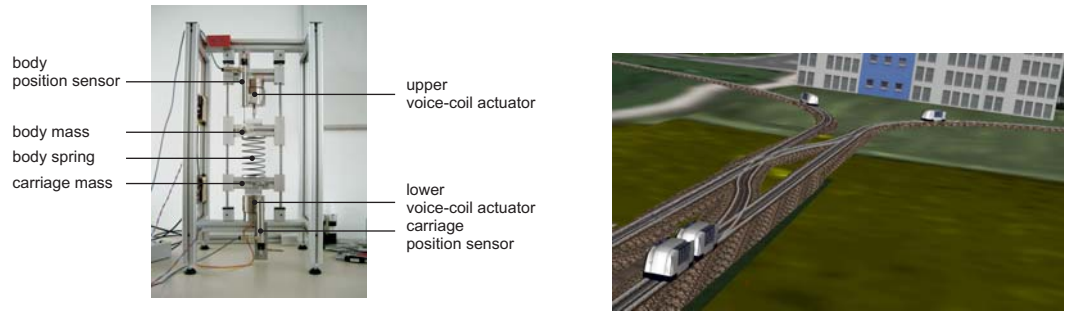
Figure 1 shows a schema of the physical model of the active vehicle suspension system and the body controller. The suspension system of railway vehicles is based on air springs which are damped actively by a displacement of their bases and three vertical hydraulic cylinders which move the bases of the air springs via an intermediate frame – the suspension frame. The vital task of the system is to provide the passengers a high comfort and to guarantee safety and stability when controlling the shuttle's coach body. In order to achieve this goal, multiple feedback controllers are applicable with different capabilities in matters of safety and comfort. The right side of the figure shows the controller model of the suspension module.

As a concrete example, we will later look into the control system of a testbed of a magnetic-levitation train (Figure 2(a)). The testbed allows a control of the body mass only in the vertical direction. It consists essentially of a supporting frame, two vertical guides for the body and carriage mass, sensors to obtain the position of the body and the carriage mass, two voice-coil actuators for the simulation of disturbances as well as the levitation-motor and the body and carriage mass connected by a mechanical spring. On the basis of this testbed we will show the modeling and code synthesis of a switchable control.

Another later considered detailed example is the control of shuttle convoys (Figure 2(b)). The 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

[1] http://nbp-www.upb.de/en/

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



(a) (b)

Figure 2: Testbed for the active vehicle suspension system (2(a)) and convoy building Shuttles (2(b))

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. Additionally, different controllers are used to control the speed of a shuttle as well as the distance between the shuttles. The controllers have to be integrated with the aforementioned real-time coordination.

Problem Statement Advanced mechatronic systems such as the outlined RailCab system require solutions which incorporate *multi-paradigm models* and exhibit *flexible reconfiguration* where the effects of reconfiguration can propagate *across module boundaries* as we will exemplify in more detail later on. For such flexible reconfigurable hybrid systems specified by multi-paradigm models, an efficient evaluation scheme is required which excludes deadlocks caused by the evaluation of the continuous parts of the hybrid system. However, existing approaches fall short in providing the required efficient solution. White-box approaches require too many global evaluation schemes and thus too much memory for reconfigurable hybrid systems. Black-box approaches in contrast result either in deadlocks during evaluation or require iterative evaluation schemes which are not appropriate for real-time processing. Schemes relying on the dataflow are usually inefficient and runtime failures cannot be excluded. Existing modular techniques like [6, 18, 19, 20, 45, 46] consider a (restricted) kind of reconfiguration but do neither consider the required propagation across module boundaries nor a flexible kind of reconfiguration. Furthermore, these approaches do not support an integration of *proper analysis techniques* in order to ensure the correct real-time behavior.

Contribution In order to apply component-based models for the specification of advanced mechatronic systems with reconfiguration, we have invented MECHATRONIC UML [16, 24] as domain–specific refinement and extension of the Unified Modeling Language (UML) [55, 56]. Among others, MECHATRONIC UML defines hybrid components[3] and hybrid reconfiguration charts [23] which permit the integrated modeling of discrete behavior specified by the hybrid reconfiguration charts in the form of extended timed automata models with continuous components (e.g., feedback controllers) that are specified by block diagrams or differential equations. In former papers we have also described in detail the techniques to ensure the correctness of the reconfiguration with respect to given real-time constraints [23, 27] and the available tool support for the approach [12]. In Table 1 the basic paradigm formalisms used in mechatronic systems that are covered by MECHATRONIC UML are presented [32].

As identified in the problem statement, complex reconfigurable hybrid systems as they result from MECHATRONIC UML models cannot always be directly addressed with existing simulation and code generation techniques. The focus of this article is therefore the modular execution scheme for reconfigurable hybrid systems which result from the combination of the different paradigms that overcome this problem

[3] MECHATRONIC UML supports different component types. We explicitly name them except for general software components which we refer to as components.

paradigm	continuous	event-based	dynamic structural adaptation
formalism	block diagrams	automata	story pattern
semantics	differential equations	timed automata	graph transformation systems

Table 1: Paradigms used in MECHATRONIC UML

by operating with rather restricted memory while remaining still efficient. We will outline its capabilities and describe in detail which analytic information is required to be able to synthesize the modular execution logic.

Our solution for handling the required complex interplay between discrete behavior and continuous behavior with a modular hierarchical execution scheme is described in this paper in detail. The ideas for this scheme have been published first in [8, 14, 23, 52]. In this paper we integrate these ideas with the formal semantics of the models, provide the complete details required to employ the modular evaluation scheme, describe the static analysis and runtime checking techniques to safeguard the development and report about the concrete tool integration efforts and the available simulation capabilities. In addition to our previous work, we extend the former results to also cover more general forms of reconfiguration without losing the benefits of the developed modular scheme where possible.

The approach has been realized for the integration of the UML tool Fujaba [4] and the CAE ((Computer Aided Engineering)) tool CAMeL-View[5] for the simulation environment IPANEMA [9]. The modular execution scheme effectively enables the integration of models of both tools by enabling a shared notion of a hybrid component using the modular execution capabilities. Finally, we also present the simulation and visualization for validation purposes for a reconfigurable hybrid model within the CAMeL-View tool.

Outline The paper is structured as follows: We first discuss the state of the art in Section 2. Then, we sketch the MECHATRONIC UML approach for modeling reconfigurable hybrid systems in Section 3 with the focus on the underlying integration of different paradigms using the introduced example. Then, we look into the problem of ordering the evaluation of reconfigurable continuous models in Section 4. We will present in detail how the required characteristics for a modular execution scheme can be derived from the models and employed to resolve this problem. An overview about support for the correct development with models employing the modular scheme by static analysis and run-time checks is sketched in Section 5 and the underlying integration of the two involved tools is presented in Section 6 referring also to the example. The paper closes with a conclusion and an outlook on future work.

2 State of the Art

The considered state of the art approaches are chosen on the basis of their capabilities of modeling hybrid systems (see Section 2.1) and their capability of generating code for continuous and hybrid systems (see Section 2.2).

2.1 Modeling of Hybrid Systems

MATLAB/Simulink and Stateflow[6] are the de facto industry standard for the modeling of technical systems. Reconfiguration can be modeled by *conditionally executed subsystems* which are triggered by control signals. Another approach for the modeling of reconfiguration is the integration of discrete blocks (Stateflow models) into block diagrams. The alternative controller outputs are fed into a discrete block whose behavior is described by a Stateflow model. Dependent on the Stateflow models' current discrete state, the corresponding continuous signals are blind out or directed to the block's output. This enables switching between the output signals of different controllers. Thus, a Stateflow model can be used to only trigger the

[4] www.fujaba.de
[5] www.ixtronics.com
[6] www.mathworks.com

required elements of the currently active configuration instead of blinding out the results of those that are not required. This approach allows modeling of reconfiguration but it has the disadvantage that systems will become very complex.

Hybrid bond graphs [50] introduce so-called *controlled junctions* to model reconfiguration. A finite-state machine (FSM) is associated with each controlled junction. Each state of the FSM is of the type on or off, indicating if the controlled junction acts like a normal junction or as a 0 value source. Therefore, state changes turn parts of the model on or off. Modeling reconfiguration with hybrid bond graphs has the same drawback like conditionally executed subsystems, as graphs consist of all active and inactive configurations.

Other approaches combining components and hybrid automata concepts such as CHARON [2] (and its extension R-CHARON [42]), HyROOM [3, 60], HyChart [28, 59], HybridUML [4], and Ptomely II [44] provide hierarchical automata models for the specification of behavior and hierarchical architectural models. In UML^h [22], the architecture is specified by extended UML class diagrams that distinguish between discrete, continuous, and hybrid classes. Also, the OMG effort to integrate models from the software engineering domain with models from the control engineering domain falls into this category. The Systems Modeling Language (SysML) [57] is a first proposal to standardize system engineering, which could be integrated with a possible UML 2.0 successor (see [40]).

All presented tools and techniques for hybrid components support the specification of a system's architecture or structure by a notion of classes or component diagrams. All approaches support modular architecture and interface descriptions of the modules. Nearly all approaches embed the continuous models in the discrete state machines using the hybrid automata concept in order to model reconfiguration of the continuous behavior within one module. Specifying reconfiguration with SysML should be possible with the activity diagrams, but concrete examples for this issue do not exist. Nevertheless, the approaches do not respect that a module can change its interface due to reconfiguration which can lead to incorrect configurations and further the approaches do not permit that reconfiguration takes place across module/ block boundaries.

CHARON, Masaccio, HybridUML with HL^3, UML^h, HyROOM, and HyCharts have a formally defined semantics, but due to the assumption of zero-execution times or zero-reaction times, most of them are not implementable, as it is not realizable to perform a state change infinitely fast on real physical machines. CHARON is the only approach providing an implementable semantics. HyCharts are implementable after defining relaxations to the temporal specifications. They respect that idealized continuous behavior is not implementable on discrete computer systems. Further, CHARON provides a semantic definition of refinement which enables model checking in principle. Ptolemy II even provides multiple semantics and supports their integration.

2.2 Composition and Evaluation Order

Given a continuous system, to evaluate the differential equation the different parts of the equation system have to be executed. For a correct execution the overall evaluation order has to be respected in order to prevent a deadlock during computation or invalid results [39]. The connections of subsystems has a direct impact on the method of the so-called model integration, i.e., embedding of subsystems into the simulation platform. In the process one has to distinguish between different kinds of integration: In a *black-box integration*, communication can disregard the inner structure of the subsystem while in a *white-box integration* the communication routines are implemented purposefully into the evaluation functions of the subsystems. A *dataflow integration* is the third alternative. And finally, we discuss *mixed approaches*, which uses the benefits of the aforementioned approaches.
The most simple solution to determine the evaluation order is to use *white-box integration*. To this end, the overall evaluation graph is built and the evaluation is done in an interwined manner such that the nodes of different blocks have to be evaluated separately. Since the sequence of the evaluation can be computed only for the entire system, this integration approach is inherently non-modular and only permits to derive optimizations concerning the granularity of the execution logic for the whole system.

White-box integration thus solves the problems concerning the evaluation order for a whole system if we have a static structure and do not require modular execution. In the formal semantics defined in Appendix A.2.2, we simply flattened the hierarchal structure and composed the directed acyclic evaluation

graphs when composing continuous blocks and thus effectively employed *white-box integration* as this is the state of the art approach by control engineers. However, while theoretically sound such a solution does (high number of evaluation orders due to the exponentially) not fit to our needs for the execution logic of reconfigurable hybrid systems.

In [49] a Java and a C/C++ export to support simulation of hybrid bond graphs is outlined. The evaluation order has to be derived for every global state (the cross product of FSMs of all controlled junctions). In our approach, we exploit the hierarchical component structure to build a tree structure, that avoids getting a number of evaluation orders that is exponential in the number of states. Further, a switch of a discrete state in a FSM can trigger transitions in other FSMs. Therefore, no upper bound is given, describing the number of transitions, which fire before the system reaches a consistent, stable state and continuous evaluation can proceed. In our approach the discrete and the continuous evaluation are decoupled and the upper bound of firing transitions is set by the hierarchy.

An alternative for determining the evaluation order which allows to use a more coarse-grain structuring of the execution logic is *black-box integration*. In our approach, only the input/output interfaces of the submodels are considered. Their inner structure remains hidden. Therefore, communication occurs only vectorially at certain, precisely defined moments in the program run [39].

In order to reduce dependencies, the equations can be sorted according to the categories non-direct link (ND), direct link (D), and state equations (S). These categories serve to classify the input/output variables [38] for $\underline{\dot{x}}$ the state vector, $\underline{y}$ the output vector, $\underline{u}$ the input vector, $\underline{p}$ the parameter vector and t the time: Non-direct links: All output variables that do not depend directly on input values are non-direct links and can thus be computed directly: $\underline{y}_N(t) = \underline{f}\left(\underline{x}, \underline{p}, t\right)$ (see y_1 of Figure 3(a)). Direct links: This category comprises all input variables that have an immediate effect on at least one output variable: $\underline{y}_D(t) = \underline{f}\left(\underline{x}, \underline{u}, \underline{p}, t\right)$ (see y_2 or y_4 of Figure 3(a)). State variables: Input variables that apply only to state equations fall into this category: $\underline{\dot{x}} = \underline{f}\left(\underline{x}, \underline{u}, \underline{p}, t\right)$ (see $\dot{x}$ of Figure 3(a)).

However, in principle, the problem of a communication deadlock due to a cycle in the evaluation graph which results from the composition of two continuous blocks cannot be fully solved just by introducing the ND-, D-, and S-blocks. A deadlock can occur, if the couplings between direct-link blocks make up a cycle. The model cannot be evaluated at the chosen granularity level any more because the subsystems required to compute the data are waiting for the data of the other subsystems. These cyclic waiting dependencies can thus result in a conceptual deadlock (Figure 3(b)).

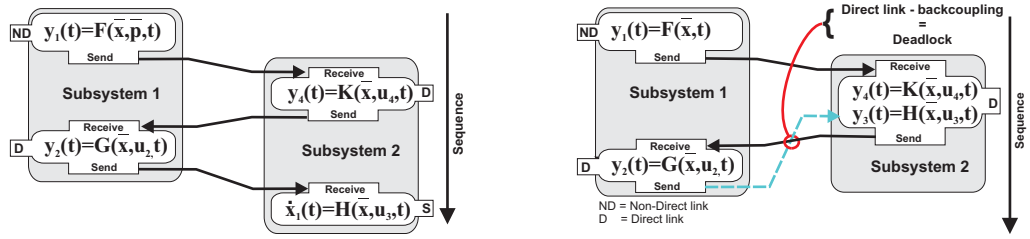


Figure 3: Correct (3(a)) and deadlocked (3(b)) black-box integration with ND-D-S decomposition

The deadlock can be dissolved by employing iterations or by white-box approaches. Descriptor methods based on differential-algebraic equations (DAE) are an approach for supporting modular simulation based on iterations. One drawback of these methods is that they cannot be used in real-time contexts because of the iterative parts of the necessary solvers. There are several CAE tools that are also based on DAEs, such as ADAMS[7] or SIMPACK[8], which allow multi-body modeling and simulation yet have the same drawbacks in real-time evaluation as do all DAE-based tools [29].

Another feasible solution is the use of filters. However, they alter the model and thus the system behaviour, which is in most cases unaaceptable [18, 19].

[7] www.adams.com
[8] www.simpack.com

Another option is *dataflow integration* which maintains no precomputed evaluation orders but use the flow of data via the expressions and connections to control the evaluation. While conceptually elegant and even applicable for any imagined reconfiguration, this solution results in two serious problems.

At first, deadlocks in the evaluation order are not detected during the compilation and composition of the system but rather at runtime [48]. This is a rather unsafe situation which is often not acceptable for many mechatronic systems as oftentimes critical functionality is realized and thus runtime failures which cannot be resolved easily at runtime also have to be excluded. Furthermore, [48] does not support reconfiguration across module boundaries.

Secondly, evaluating each single equation related to the nodes of the evaluation graph of the system as a single unit results in an extremely slow execution logic. In between each single assignment the potentially complex processing of the dataflow and the availability of all inputs of an equation happens.

Therefore, this alternative is in fact no real option for an industrial strength solution for reconfigurable hybrid systems unless the models executed under this regime are rather small. However, we look for a solution for large, complex mechatronic systems.

It is also possible to avoid such runtime checks and rely on computing a fixpoint at runtime [21, 43]. However, this fixpoint may contain undefined values, which means such code cannot be used in safety-critical, hard real-time applications.

The last solutions are approaches, which mix the different other approaches. Approaches for modular code generation of synchronous languages, like MATLAB/Simulink and LUSTRE [17] or Esterel [5], have been presented in [6] and [45, 46] . Modular code generation is mandatory for synchronous languages to find adequate variables which lead to deadlock free and efficient code. These approaches abstract each block in a way which enables modular code generation. In [45, 46] for example, a set of interface functions of each block and a set of dependencies between these interfaces is generated. The finer the abstraction, the more information about the input – and output dependencies is preserved, which enables the block to be more reusable. This is similar to our grey box approach for the evaluation of block diagrams (see Section 4), but these approaches did not consider reconfiguration across module boundaries as well as no flexible reconfiguration as requird in our problem statement (see Introduction).

In [18, 19, 20], synchronous languages with dynamic reconfiguration are considered. The authors present a conservative extension of LUSTRE with hierarchical state automata, based on a translation semantics into a clocked data-flow kernel. The authors advocate that such a translation not only gives the semantics of the whole language, but is an effective way to implement the compiler (code generation) in the sense that the generated code is efficient and of small size. In this approach cyclic dependencies are avoided by requiring that every feedback loop is broken by a unit-delay block at every level of the hierarchy. This is a major modeling restriction, as most diagrams in practice exhibit feedback loops with no such delays at higher levels of the hierarchy.

2.3 Comparison

In summary, the existing approaches fall short when it comes to reconfiguration that is not restricted to effects within a single module. In addition, existing evaluation schemes are not able to cope with the complexity when evaluating reconfigurable hierarchical hybrid systems. The presented solution for MECHATRONIC UML overcomes these limitations providing modeling capabilities as well as an efficient evaluation scheme for the resulting complex models.

3 Multi-Paradigm Modeling

In this section, we consider how the different modeling paradigms for the control and software engineering domain are integrated in the MECHATRONIC UML approach (see Table 2). We will especially focus on the resulting support for reconfigurable hybrid systems.[9] We introduce here only the supported notation. A

[9] Note that besides providing a solution to the multi-paradigm modeling problem, in practice rather than simply a new language support for the different established domain-specific notations and tools is needed. The Fujaba Real-Time Tool Suite presented in Section 6 therefore realizes the MECHATRONIC UML approach by integrating an existing CAE tool.

detailed formalization of the concepts including the basic syntax and semantics is presented in Appendix A.2.

paradigm	continuous	continuous + event-based	dynamic structural adaptation
formalism	block diagrams	hybrid reconfiguration charts + hybrid components	story pattern + hybrid components
semantics	differential equations	hybrid reconfiguration automata	hybrid graph transformation systems
tool	CAMeL	Fujaba	Fujaba

Table 2: Paradigms used in MECHATRONIC UML

3.1 Continuous Models

As stated in the introduction, a common technique for the specification of controllers that is widely-used in different tools is the notion of hierarchical block diagrams.

Block diagrams generally consist of basic blocks, specifying behavior, and hierarchy blocks that group basic and other hierarchy blocks to reduce the visual complexity. Each block has input and output signals. The unidirectional interconnections between the blocks describe the transfer of information. For example, the output signal of a basic block is fed as an input signal into a hierarchy block.

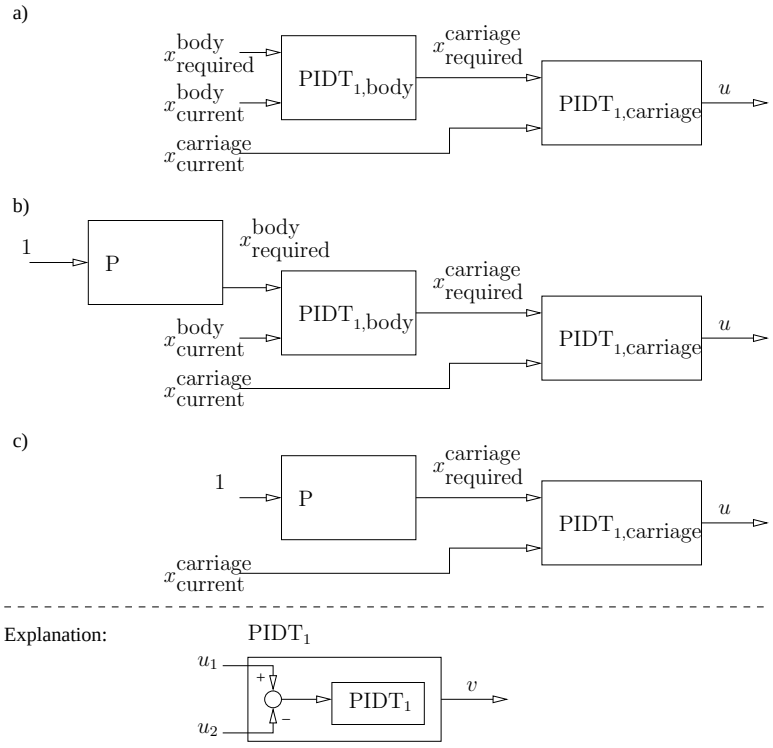


Figure 4: a) Comfort Controller, b) Semi-Comfort Controller, and c) Robust Controller

Figure 4 displays the three controllers that are applied in our example. In Figure 4 the *Comfort* controller, providing the passengers the most comfort, is shown. It consists of two PIDT$_1$[10] controllers – one

[10]PIDT: Proportional-integral-derivative-time controller

for controlling the undercarriage, the other for controlling the coach body. For inputs, PIDT $_{1,body}$ obtains the desired and the actual positions of the coach body. The first one is provided by a user input, the latter by a sensor. The output yields the position of the undercarriage and serves as an input for PIDT $_{1,carriage}$. The other input, the current position of the undercarriage, is provided by a sensor as well.

If, however, a user input does not exist, this value is set to a constant value as displayed in Figure 4b. In case the $x_{current}^{body}$ sensor should fail, this controller structure could lead to an instability. Then the system needs to be reconfigured as shown in Figure 4c. The required position of the undercarriage is set to a constant value. This controller provides less comfort, but guarantees stability. To ensure stability, fault tolerance patterns are applied in such a way that the signal $x_{current}^{carriage}$ is computed redundantly such we can rely on $x_{current}^{carriage}$ even in the case on sensor data fails.

A single block is characterized by input, output and auxiliary variables and a set of expressions with a left-hand side variable and a right-hand side expression with references to other variables (see Figure 5).

```
Inputs: U1, U2, U3, U4
Outputs: Y1, Y2, Y3, Y4
States: X=0;
Auxilars: a, b, c, d

a  := U1 + U2;
b  := U3;
c  := U4;
d  := 1;
X' := X + c;
Y1 := a;
Y2 := b;
Y3 := b;
Y4 := d*x;
```

Figure 5: A basic continuous block

If the set of equations is well-formed (see Appendix A.2.1), it can be represented by a corresponding directed acyclic evaluation graph. Each left-hand side variable is represented by a node and all occurrences of variables in the right-hand side as an edge from the node of the referenced variable to the node of the defined variable. Following this outline, we can derive an acyclic evaluation graph $G = (N, E)$ with node set N and edge set $E \subseteq N \times N$. For each $n \in N$ and the related expression $v := \ldots v' \ldots$ holds that for each variable v' the expression refers to an edge $(n', n) \in E$ with n' is related to v'. In addition, we have for each evaluation graph $N_{state} \subseteq N$ denoting the subset of nodes which represent the internal state of the block. In the case of a well-formed composition of two continuous blocks, we can derive the resulting directed acyclic evaluation graph by simply combining the graphs of both blocks at the connected inputs and outputs. In the case of a single block as well as a composition, a simple check of the evaluation graph at compile-time is sufficient to exclude problems with the evaluation at run-time.

3.2 Reconfigurable Hierarchical Hybrid Models

The architecture of the system is based on distributed, interconnected components. The components are based on UML 2.0 components but we distinguish between different types of components (see Figure 28). We distinguish between (discrete) components, hybrid components, and continuous components (block diagrams). As required in the introduction of Section 3, components can embed other components. Additionally, we can distinguish between component types and component instances.

Figure 6 shows an instance view of the component structure of the suspension system. There it is shown that a hybrid Monitor component embeds continuous Sensor and Storage components and moreover a hybrid BodyControl component, which embeds three different controllers (not shown in the Figure). Based on the availability of the information of the Storage, which stores track information of other shuttles, which are communicated from the registry and the Sensor of the BodyControl can switch between different controllers based on the available information. Besides the shown embedding of the different paradigms, a more abstract view in the form of patterns are also supported as shown in the Figure. A pattern in our

case consists of port roles (MonitorRole, RegistryRole) and a connection between the roles. Besides the structure, a pattern consists also of a behavioral description [27].

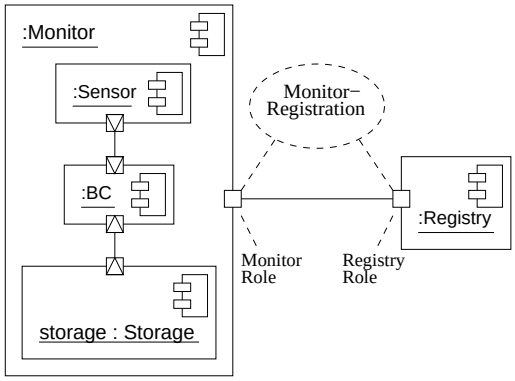


Figure 6: Structural description of the suspension system

The behavior of this hybrid BC component is specified by a simple Hybrid Statechart (see Figure 7) which is an extension of a Real-Time Statechart [7]. It consists of three different control modes (discrete states) associated with the three configurations from Figure 4.

The configurations consist of PIDT$_{1,carriage}$, PIDT$_{1,body}$, and the P blocks. For switching between two controllers, we can distinguish between atomic switching and cross-fading. If the switching between two blocks can take place between two computation steps, atomic switching is used and otherwise cross-fading. The cross-fading itself is specified by a fading function and an additional parameter which determines the duration of the cross-fading. Bold arrows indicate that output cross-fading, which consumes time, has to be applied when there is a switch between two states. The deadline intervals d_i specify the minimum and maximum fading time allowed. In contrast, thin arrows indicate that a switch is performed without time-consuming fading.

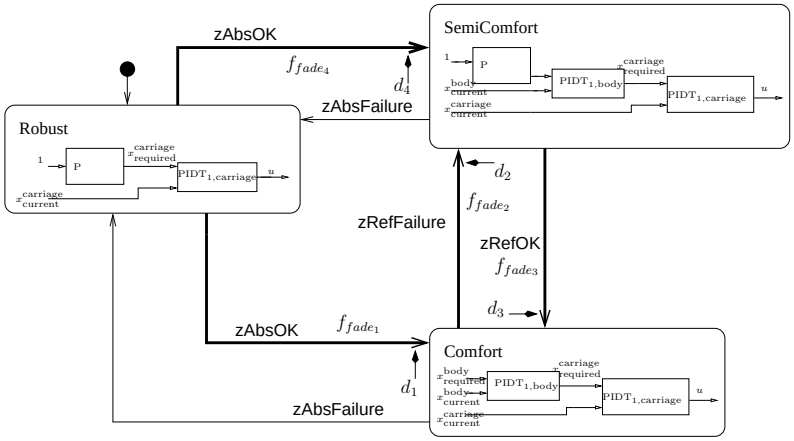


Figure 7: Behavior of the Body Control component

When using this component in advanced contexts (e.g., embedding the component in another configuration), usually an abstract view of the component without the implementation details is sufficient. This view is given by the hybrid *Interface Statechart* (see Appendix A.2) of the component (see Figure 8). It consists of the externally relevant real-time information (discrete states, their continuous in- and outputs, possible state changes, their durations, signals to initiate transitions, and signal flow information [53]). They abstract from the embedded components and from the fading-functions. Ports that are required in each of the three interfaces are filled in black, the ones that are only used in a subset of the states are filled in white.

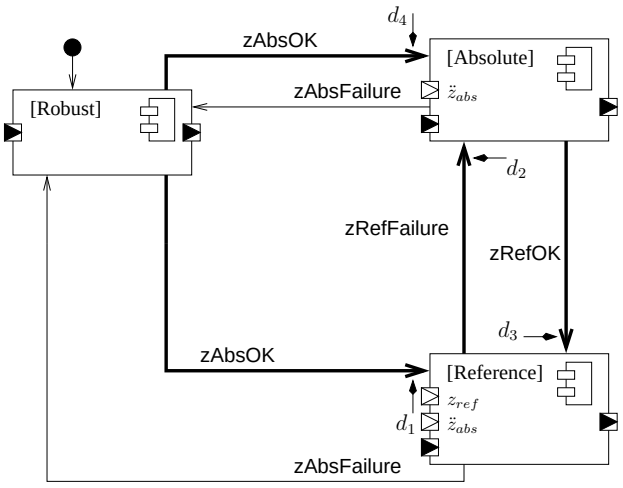


Figure 8: Interface Statechart of the Body Control component

In this example, each discrete state of the component has a different continuous interface, a fact that leads to an Interface Statechart which consists of as many discrete states as does the detailed Hybrid Statechart. Usually, not every internal reconfiguration will result in a different external state, which leads to further reduction of complexity in the Interface Statechart. The external view of the monitor component (see Figure 10) does not change at runtime although it consists of four discrete states. Therefore, its Interface Statechart consists of just one state.

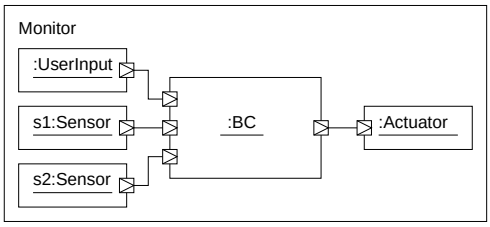


Figure 9: Structure of the controlling system

As displayed in Figure 9, the BC component is structurally embedded into the Monitor component. Figure 10 shows the behavior of the monitor embedding and coordinating the behavior of BC and the other embedded components as described in [14, 23]. For this embedding the notation from Interface Statechart (see Figure 8) is used.

The Hybrid Statechart in Figure 10 consists of four states representing that (i) a user input exists as well as the sensor providing $x^{body}_{current}$ (state AllAvailable), (ii) both of these signals are not available (state NoneAvailable) and (see (iii) and (iv)) exactly one of these signals is available (states BodyAvailable and UIAvailable). Note again that the sensor providing $x^{carriage}_{current}$ is laid out as fault-tolerant.

Every state is associated with a configuration. Thus, a switch from NoneAvailable to BodyAvailable results in a reconfiguration and a switch of the embedded BC component from state Robust to SemiComfort. The states UIAvailable and NoneAvailable are required to keep track of the available signals, yet a switch between them will not lead to a reconfiguration or further state switches.

Transitions, visualized by bold arrows, are associated with deadlines because they are time-consuming. They are triggered by events, raised from the xBody Sensor or UserInput component.

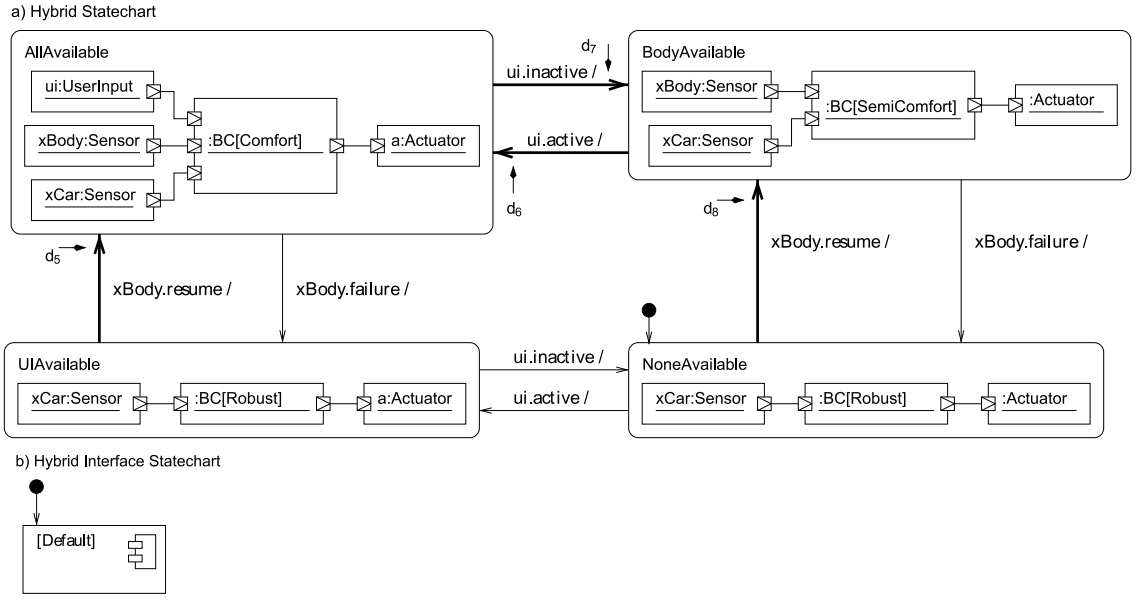


Figure 10: a) Behavior and b) Interface Statechart of the Monitor component

3.3 Flexible Reconfigurable Hierarchical Hybrid Models

The approach presented so far can effectively be applied when the required reconfiguration is local. Usually, all possible configurations are well-known at the design time and their number is small. However, specifying more flexible reconfiguration which results from the need to coordinate ad hoc groups cannot be addressed.

When shuttles build a convoy and a leader shuttle determines the reference positions for all the following shuttles, the control of these reference positions depends on the length of the convoy and on the participating shuttle types and characteristics. For example, a heavy load shuttle has to hold a larger distance within the convoy. The leader shuttle of a convoy can respect such individual properties or requirements only when individual components or feedback-controllers are applied to determine the reference positions. Using our approach presented so far would thus be impractical as a large number of possible configurations (in principle even infinite many ones) have to be explicitly specified.

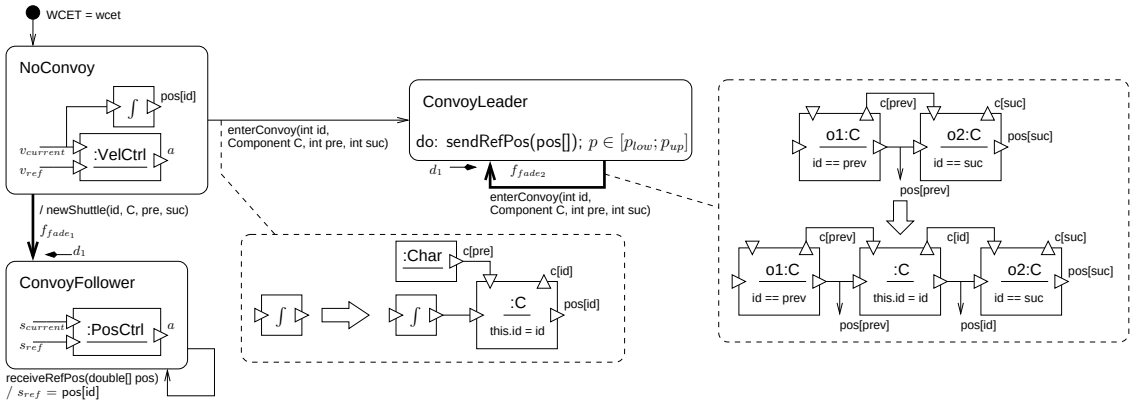


Figure 11: Shuttle behavior to control convoys

In the given example, the different shuttle types are not known a priori at design time (recall how many different types of automobiles exist). Thus, each shuttle sends the component, which the leader shuttle has to apply, to the leader shuttle when it joins the convoy at runtime. We therefore suggest to specify

the required flexible structural reconfiguration by means of *reconfiguration rules* where control elements can be determined by parameters which are based on a story pattern [41]. Story patterns are based on the theory of graph transformations systems which are usually applied for model transformations. We will exemplify that they are an appropriate visual, model-based description technique for the specification of reconfiguration at runtime.

A cut-out of the behavior of the shuttles for *coordinating* convoys is depicted in Figure 11. In [27], we describe how to ensure a safe *building* of convoys. The hybrid reconfiguration chart consists of three states: ConvoyLeader represents that the shuttle is the leader shuttle, ConvoyFollower represents that the shuttle is part of a convoy but not the leader shuttle, and NoConvoy represents that the shuttle is not in a convoy at all.

Residing in state ConvoyFollower, the shuttle applies a position controller that delivers the current acceleration a dependent on its reference position s_{ref} and its current position $s_{current}$. It periodically receives the event receiveRefPos with parameter pos[] and stores the new reference position pos[id] as a side-effect in s_{ref}. In state NoConvoy, the shuttle applies a velocity controller, requiring a reference and the current velocity as input. The latter one is used to determine the current position pos[id]. When a new shuttle joins the convoy, it sends an event enterConvoy with the following parameters: its identifier id, the component C to be used to determine the shuttle's reference position, and the IDs pre and suc of the shuttles which let the new shuttle in.

The reconfiguration rule of the transition (visualized with a dashed border) adds the component C to the shuttle's control structure: An instance of a component Char is created that provides the characteristics of the leader shuttle such as length, maximal brake acceleration. These characteristics and the current position of the leader shuttle are fed into C which determines the reference position pos[id] as output. A simple implementation of component C would just add the length of the preceding shuttle and an individual safety margin to the current position of the preceding shuttle. Port c[id] provides the characteristics of the new shuttle.

Residing in state ConvoyLeader, the shuttle sends periodically with a period $p \in [p_{low}; p_{up}]$ the reference positions pos[] to the according shuttles. If a further shuttle joins the convoy, its component is inserted in the structure between the components of prev and suc. Reconfiguration rules for the special cases when a shuttle joins at the end or at the beginning of the convoy or for the case when a shuttle leaves the convoy are omitted in Figure 11. Due to lack of space, we omitted also transitions which model that shuttles leave the convoy. If we specified a transition, leading from ConvoyLeader to NoConvoy, the current configuration –eventually consisting of multiple components– would be discarded and the configuration of NoConvoy would be applied.

The example points out that modeling flexible reconfiguration with reconfiguration rules leads to an enormous reduction of the visual complexity, as not every possible configuration has to be specified explicitly.

4 Modular Execution

In the following, we will present in detail our concept for the modular execution of reconfigurable hybrid systems for the reconfigurable models sketched in the former section. First, we discuss our solution for modular hierarchical continuous systems and then extend it to reconfigurable hierarchial hybrid systems. Finally, our solution for the continuous dataflow part of flexible reconfigurable hierarchical hybrid systems is presented. For completeness, the syntactic and semantic foundation are presented in Appendix A.2.

4.1 Modular Composition and Evaluation Order

For providing a feasible evaluation scheme for the hierarchical reconfigurable hybrid systems presented in the last section, we require as a crucial prerequisite a modular composition scheme which is less restrictive and more efficient than black-box integration (see [45]). Our proposed approach [8, 52] outlined in the following will combine the advantages of both the white-box and the black-box approach and is thus named *grey-box integration* (originally coined in [54] for the dynamic computation of the evaluation order at runtime).

With a grey-box method one tries to combine the advantages of the white-box with those of the black-box. The sequence of evaluation for grey-box integration is determined in two stages. The first stage defines the local sequence of evaluation of each basic block independently of an external coupling. For a hierarchy block, the coupling information on its children and the resulting reduced evaluation graph will be determined and further employed. This implies an independent derivation of the execution logic for every block; thus a model can be generated from independent modules. At the second stage the external coupling information yielded by the hierarchy blocks is taken into account for determining the entire sequence of evaluation and making up the evaluation graph. This presupposes a well-defined module interface. The information required for this step were obtained in the preceding stage so that this procedure can be completed during initialisation of the application.

When proceeding in this way we are able to hide the internal logic of a module. This is especially important if one wants to exchange models and integrate them even though their content is to be kept secret. The grey-box method solves the problem of the direct-link feedback coupling by further decomposing the equations in the direct-link block into independent partial blocks. Appurtenance of the equations to a partial block is defined by their dependence on one or several input and output variables, with an input or an output variable being allocated definitely to just one partial block in the direct-link block.

In order to derive the required information to embed the acyclic evaluation graph in an arbitrary context, we have to partition it into separate *evaluation blocks* (see Figure 12, right). For the optional decomposition into such evaluation blocks, it must hold that when the original evaluation graph $G = (N, E)$ can be safely embedded in a given context that also the evaluation blocks condensed evaluation graph can be safely embedded.

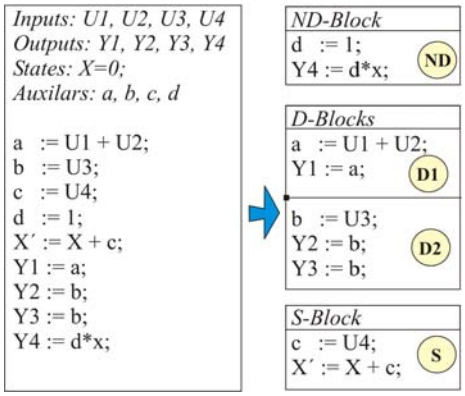


Figure 12: Partitioning of the evaluation graph

The *partitioning problem* for a given acyclic expression graph $G = (N, E)$ of a continuous block is thus to determine a minimal number of partitions $N_1, \ldots, N_n \subseteq N$ such that:

1. $N = N_1 \cup \cdots \cup N_n$ and $\forall i \neq j \; N_i \cap N_j = \emptyset$,
2. the derived graph $G_p = (N_p, E_p)$ with $N_p = \{N_1, \ldots, N_n\}$ and $E_p = \{(N_i, N_j) | i \neq j \wedge i, j = 1, \ldots, n \wedge \exists n' \in N_i \wedge n'' \in N_j : (n', n'') \in E\}$ is acyclic, and
3. for any context graph $G' = (N', E')$ with $(N \cap N') \subseteq (N_{in} \cup N_{out})$ and $G'' = (N'', E'')$ with $N'' = N \cup N'$ and $E'' = E \cup E'$ an acyclic graph holds that the related derived graph for the partitioning built by $N_1, \ldots, N_n$ and each node of $N' - N$ is also an acyclic graph.

Condition 1 ensures that the partition sets N_i cover the full node set N. The partitioning preserves the acyclic nature (see enumeration 2). Finally, condition 3 informally states that the chosen partitioning when embedded into an arbitrary context only results in a cycle when also the original graph would also result in a cycle.

For a given acyclic graph $G = (N, E)$, we can compute the required maximal partitioning of such a graph in several phases:

First, we compute the set D_{in} of all inputs the node depends on in a forward topological traversal of the graph. Then, we do a backward traversal to compute all influenced outputs I_{out} for all input nodes. Based on D_{in} and I_{out} we can then determine the nodes of the S-block and ND-block. The remaining nodes are finally partitioned by propagating the dependencies L. Two nodes n and n' are then assigned to the same partition ($N_{L[n]}$) iff $L[n] = L[n']$. The resulting algorithm is presented in Figure 13.

```
block_partitioning((N,E)) begin

  // input dependencies
  D_in : N → ℘(N_in ∪ N_state);
  // influenced outputs
  I_out : N → ℘(N_out ∪ N_state);
  // related input nodes
  L : N → ℘(N_in);
  // visited successors
  c : N → ℕ;

  // forward traversal to compute
  // all input dependencies
  F := N_in ∪ N_state;
  forall n ∈ F do D_in[n] := { n }; done
  forall n ∈ N do c[n] := 0; done

  while ( F ≠ ∅ ) do
    forall (n ∈ F ) do
      forall n' ∈ N with (n,n') ∈ E do
        c[n'] := c[n'] + 1;
        if c[n'] == d_in((N, E), n') then
          D_in[n'] := ⋃_{(n'',n')∈ E} D_in[n''];
          if (n' ∉ N_state ∪ N_out) then
            F := F ∪ {n'};
          fi
        fi
      done
      F := F − {n};
    done
  done

  // backward traversal to compute
  // all influenced outputs
  F := N_out ∪ N_state;
  forall n ∈ F do I_out[n] := { n }; done
  forall n ∈ N do c[n] := 0; done

  while ( F ≠ ∅ ) do
    forall n ∈ F do
      forall n' ∈ N with (n',n) ∈ E do
        c[n'] := c[n'] + 1;
        if c[n'] == d_out((N, E), n') then
          I_out[n'] := ⋃_{(n',n'')∈ E} I_out[n''];
          if (n' ∉ N_state ∪ N_in) then
            F := F ∪ {n'};
          fi
        fi
      done
      F := F − {n};
    done
  done

  // compute S and ND blocks
  N' := N; //nodes of the D blocks
  forall n ∈ N do
    // n is an element of the S-block
    if (I_out[n] ⊆ N_state) then
      L[n] := 'S';
      N' := N' − {n};
    else
      // n is an element of the ND-block
      if (D_in[n] ⊆ N_state) then
        L[n] := 'ND';
        N' := N' − {n};
      fi
    fi
  done

  // backward traversal to compute L
  // L[n] set of related input nodes
  E' := E ∩ (N' × N');
  F := N_out ∩ N';
  forall n ∈ F do L[n] := D_in[n] ∩ N'; done
  forall n ∈ N do c[n] := 0; done

  while ( F ≠ ∅ ) do
    forall n ∈ F do
      forall n' ∈ N' with (n',n) ∈ E' do
        c[n'] := c[n'] + 1;
        if c[n'] == d_out((N', E'),n') then
          L[n'] := ⋂_{(n',n'')∈ E'} L[n''];
          F := F ∪ {n'};
        fi
      done
      F := F − {n};
    done
  done

  // n and n' are in the same block
  // iff L[n] = L[n']
  return (L);
end
```

Figure 13: Block partitioning algorithm

4.2 Hierarchical Composition and Evaluation Order

The presented scheme is sufficient to handle the composition of a number of continuous blocks. However, systems in practice are structured by hierarchies describing the coupling between blocks and between hierarchies (Figure 14).

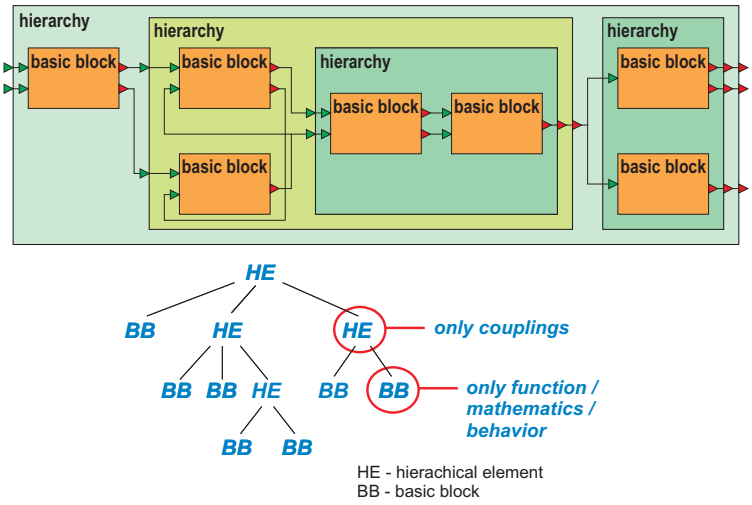


Figure 14: Hierarchical block diagrams

Figure 15 depicts our decomposition approach which enables us to structure mechatronic systems using only basic blocks (BB) and hierarchical blocks (HB). To achieve such a decomposition in a modular fashion we have to determine if it is possible to find a partitioning of the evaluation code and interface definition for modules such that the internal evaluation order of the modules can be adapted to any possible external coupling.

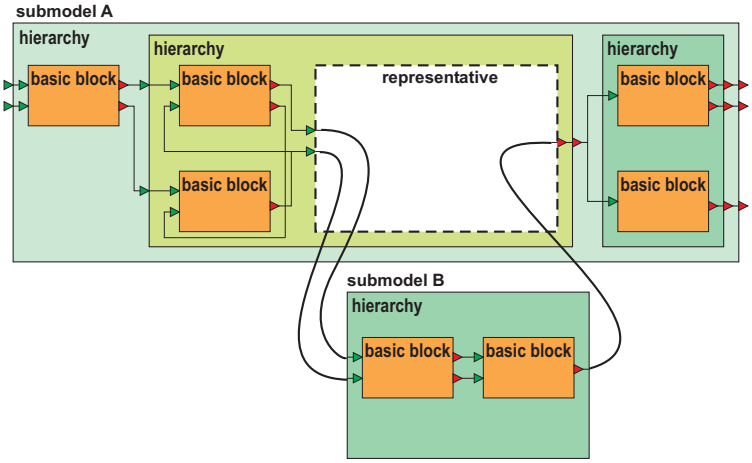


Figure 15: Decomposition of block diagrams

To extend our scheme also to hierarchical systems, we suggest to compute an abstraction in the form of a *reduced evaluation graph* which serves as an interface for the execution logic and evaluation.

To derive the reduced evaluation graph which relates to the computed optimal partitioning (see Figure 16, right), we can use the computed dependencies captured by L.

The algorithm performing this task is presented in Figure 17.

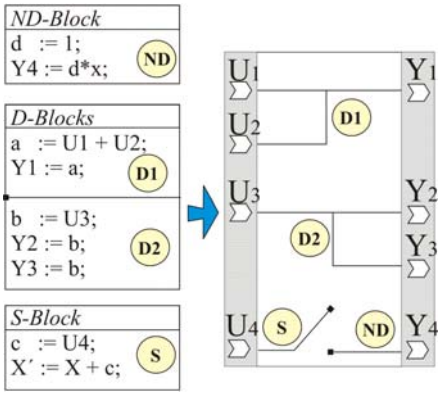


Figure 16: Derive the reduced evaluation graph for a basic block

```
reduced_evaluation_graph((N,E),L) begin
  G := ∅ // set of blocks

  // compute the S block
  F := N ∩ N_in // only consider inputs
  N_S := ∅
  forall (n ∈ F) do
    if (L[n] = 'S') then
      N_S := N_S ∪ {n}
      N := N \ {n}
    fi
  done
  G := G ∪ {(N_S ∪ {n_S}, { (n, n_ND)|n ∈ N_S})}

  // compute the ND block
  F := N ∩ N_out // only consider outputs
  N_ND := ∅
  forall (n ∈ F) do
    if (L[n] = 'ND') then
      N_ND := N_ND ∪ {n}
      N := N \ {n}
    fi
  done
  G := G ∪ {(N_ND ∪ {n_ND}, { (n_ND, n)|n ∈ N_ND})}

  // compute other blocks
  F := N ∩ (N_in ∪ N_out) // only consider outputs
  while (F ≠ ∅) do
    choose n ∈ F
    l := L[n]
    N' := {n}
    N := N \ {n}
    forall (n ∈ F \ {n}) do
      if (L[n] = l) then
        N' := N' ∪ {n}
        N := N \ {n}
      fi
    done
    G := G ∪ {N'}
    G := G ∪ {(N', { (n, n')|n ∈ N' ∩ N_in, n' ∈ N' ∩ N_out})}
  done

  return (G);
end
```

Figure 17: Algorithm to compute the reduced evaluation graph

Using the evaluation blocks as evaluation steps, it becomes possible to execute an entire system in a modular manner using the following three cases:

(1) *Local Partitioning and Local Execution Order*: For a non-hierarchical grey-box component we can compute a minimum partitioning by means of the algorithm outlined. Condition 3 then ensures that for any possible context, a correct evaluation order of the blocks can always be identified if it can be found in the white-box scenario. The subgraphs of the acyclic expression graph related to each block are also acyclic and correspond to a partial order of the expression evaluations. The partial order can be refined into a total order which can then be used to sequentially evaluate the expressions within the code for those blocks.

(2) *Hierarchical Partitioning and Local Execution Order*: The block partitioning algorithm outlined can be used to derive also the required partitioning information in a hierarchical manner. The evaluation graph which results from the reduced evaluation graphs of all embedded components is sufficient to derive again a minimal set of hierarchical evaluation blocks (see Figure 18). In contrast to the non-hierarchical blocks, these blocks consist of the blocks of the embedded components and the coupling. If no block partitioning can be found, a logical flaw in the component structure exists which would also prevent execution in the white-box scenario. If a block partitioning has been found, we also know that the original is acyclic. The subgraphs of the dependency graph which is assigned to each block will thus always be acyclic, too, and can be used to derive the required internal order of the subordinated blocks, couplings, or expression evaluations.

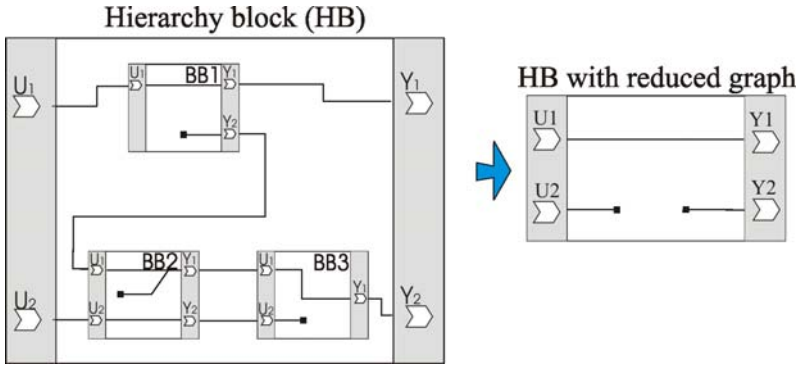


Figure 18: Evaluation graph of a hierarchy

(3) *Global Execution Order*: When the top hierarchical level has been reached, the only remaining inputs and outputs are sensors or actuators. We can therefore be sure that the resulting evaluation graphs are acyclic and we can derive the main execution logic following the scheme outlined for the hierarchical case.

To summarize, we can derive the execution logic for each component in a modular manner by (a) deriving the blocks, their local execution logic, and a reduced evaluation graph for all basic blocks, (b) deriving the blocks, their local execution logic, and a reduced evaluation graph for all hierarchical blocks considering only the internal coupling and the reduced evaluation graph of all embedded blocks, and (c) using a simple execution logic for the top-level block which only triggers the evaluation of it sub-blocks.

In the case of a static hierarchical continuous system this modular procedure has a higher overhead produced by the granularity of the partitioned blocks than a monolithic solution. However, in the next subsection we will see that the presented scheme is required to have a feasible scheme to evaluate reconfigurable hybrid systems.

4.3 Reconfigurable Hierarchical Hybrid Systems

In addition to the hierarchy, if we consider that the switching between different states of a Hybrid Reconfiguration Chart results in a reconfiguration, we have to further extend our evaluation scheme. In contrast to the static case the local state changes result in a different configuration of embedded components and their wiring which must be executed (see Figure 19).

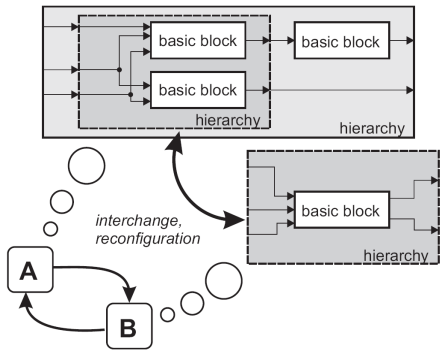


Figure 19: Decomposition of hierarchical hybrid component diagrams

For such a reconfigurable hierarchical hybrid system, a white-box integration is not feasible as we require one global evaluation order for potentially exponential many configurations. The modular scheme outlined before is, however, still applicable and can be used to realize the execution logic with low effort.

For reconfigurable hybrid systems we have extended the modular evaluation scheme presented so far as follows: Due to the additional discrete part, we have to locally determine the ordering and selection of evaluated sub-blocks depending on a local state for each hierarchical block leading to a modular scheme for hierarchical hybrid systems. Therefore, we have to derive for every discrete state the blocks, their local execution logic, and a reduced evaluation graph taking only the internal coupling and the reduced evaluation graph of all embedded blocks in the embedded mode into account.
To explain the resulting overall modular execution scheme, we outline here the details for our initial testbed. When the Monitor component is evaluated, it will trigger the evaluation of its embedded components. As not every embedded component belongs to every configuration, it depends on the current discrete state of the Monitor which of the embedded components are evaluated. Then, the triggered components will themselves trigger their embedded components (in dependency of their discrete states).

Thus, there is one *evaluation order* per discrete global state. Enhancing the top-level monitor component with this information is usually not feasible as the number of global states grows exponentially with the number of components. Therefore, we compose the whole system as a tree structure consisting of modular hybrid components.

Each hybrid component is partitioned into multiple discrete and continuous evaluation nodes using the former outlined hierarchical partitioning. The continuous evaluation nodes compute continuous states and outputs of the feedback controllers like the blocks discussed before. The discrete evaluation nodes switch the discrete states of the components and reconfigure subordinated hybrid components. As the application of certain continuous evaluation nodes depends on the actual configuration and thus on the actual discrete state, safety can only be guaranteed if one continuous evaluation cycle is not preempted by the evaluation of a discrete node which switches the actual discrete state. Therefore, we separate the evaluation of the discrete nodes from the evaluation of the continuous nodes in time.

Basic Continuous Components We have to determine the *evaluation order* of the continuous nodes in dependency of their external couplings. In order to minimize computational effort, we partition multiple single evaluation nodes of subordinated components into evaluation nodes of the superordinated component. The order within such a superordinated evaluation node is static and thus does not change. We apply the partitioning algorithm outlined in the last subsection.

The output of the algorithm is called the *reduced evaluation graph* of the configuration. It shows the interface of the component, the partitioned evaluation nodes, and their input-output dependencies. Figure 20a shows the P[11] and a $PIDT_1$ controller. Each node represents a set of expressions (e.g., mathematical expressions) and the arrows indicate the dependencies. The reduced evaluation graphs, which are obtained from the application of the partitioning algorithm, are shown in Figure 20b. The P controller consists of

[11] P: Proportional controller

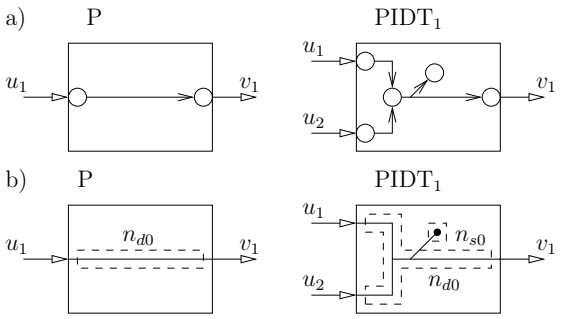


Figure 20: a) Evaluation node structure and b) *Reduced evaluation graph* of a P and a PIDT $_1$ controller

one (direct link) node n_{d0}, the PIDT$_1$ controller consists of one (direct link) node n_{d0} and a (state) node n_{s0}.

In order to evaluate the nodes in a different order, each basic continuous component is implemented as a class, providing an evalCont(int nodeId) method. According to the parameter of the method, the corresponding node is evaluated (see Figure 21). The possible nodeIds n_{d0} or n_{d0}, and n_{s0} respectively correlate to the nodes which have been determined by the partitioning algorithm (see Figure 20b), the internals (auxiliars[0]=...) are the expressions of the components (see Figure 20a).

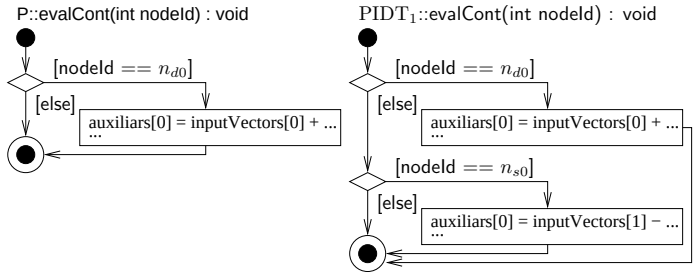


Figure 21: Activity Diagram of the evalCont(int nodeId) method of a P and a PIDT $_1$ controller

Basic Discrete Components Discrete evaluation nodes do not need to be partitioned, because every discrete component consists of exactly one discrete node. In dependency of the current discrete state, transitions are checked for activation and – as the case may be – are fired in this evaluation node.

Similar to the implementation of a basic continuous component, a discrete component consists of a method evalDiscrete(). It has no nodeId parameter as it consists of exactly one evaluation node. Furthermore, it consists of an attribute current indicating the current discrete state.[12] Inside the evalDiscrete() method, there is a check – in dependency of the current discrete state – if transitions are triggered. In case of some triggered transitions, one is selected, its side-effects are executed and the discrete state is changed. This application flow is displayed in Figure 22.

Hybrid Hierarchical Components When the controller is embedded into a well-known configuration (that defines the external coupling of the single controllers), the algorithm is applied again to partition the configuration. The algorithm does not consider the single algebraic expressions, but just the nodes from the reduced evaluation graphs of the single component.

Figure 23a shows the configuration of the SemiComfort state from Figure 7 (see Figure 4b). The partitioning results in the reduced evaluation graph of Figure 23b. Partitioning of the other configurations

[12] In the case of a flat automata model, current is usually of a simple data type, like int. In a statechart model with hierarchical and orthogonal states it is of a more complex type.

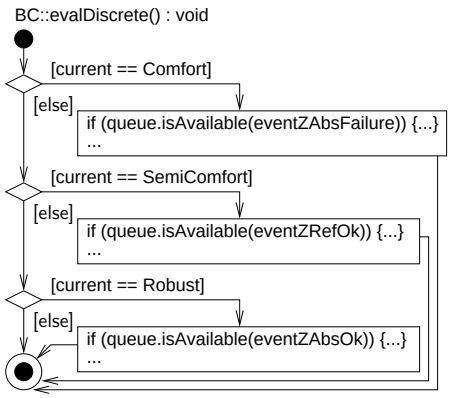


Figure 22: Activity Diagram of the evalDiscrete() method of BC

of Figure 7 (see Figures 4a and 4c) is done similarly. Thus, each of the three discrete states of the BC component is associated with another reduced evaluation graph.

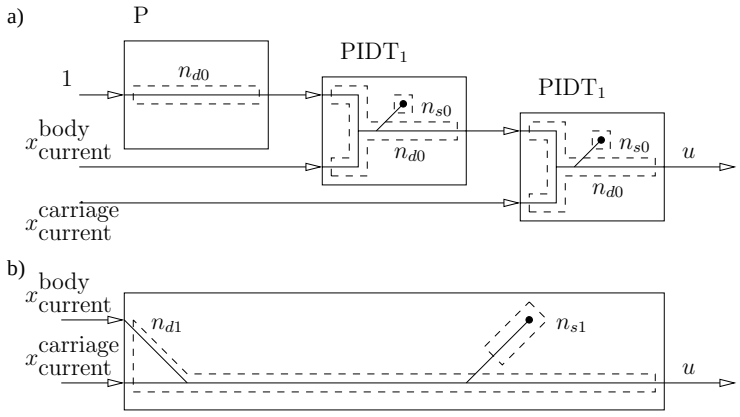


Figure 23: Configuration and reduced evaluation graph of the semi-comfort controller

As shown in Figure 10, the configuration that is associated with a discrete state does not have to consist exclusively of continuous components, but may also consist of Hybrid Components. These Hybrid Components are in a specific discrete state in this configuration. Then the partitioning algorithm works on the reduced evaluation graphs of the appropriate discrete states.

When evaluating the discrete evaluation nodes of hierarchical components, we have to make sure that a change at the top-level component affects the subordinated components within the same execution cycle. Therefore, the discrete nodes of the components at a higher hierarchy level must be evaluated prior to the ones at the lower hierarchy levels.

An implementation of the hybrid hierarchical components, like BC, consists of the methods evalCont(int nodeId) and evalDiscrete(). These methods call the according methods of the embedded components. Therefore, the hierarchical component needs to have references to these components. Figure 24 displays this structure by a UML object diagram for the BC component.

The implementation of the evalCont(int nodeId) method of a hybrid hierarchical component differs from that of a basic continuous component. It does not contain the direct code but sequences of function calls to evaluate the continuous nodes of the *embedded* (basic continuous or hybrid hierarchical) components. These lists are dependent on the current discrete state. Figure 25 shows the content of the evalCont(int nodeId) method of BC as an activity diagram. In dependency of the current discrete state each node (n_{d1} and n_{s1}) consists of a different list of evaluation nodes of embedded components. If, for example, the component is in state SemiComfort, the node n_{d1} will consist of three sequential calls of the nodes n_{d0}

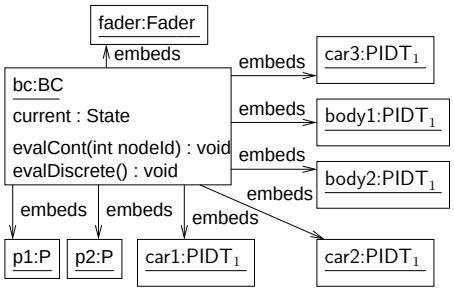


Figure 24: Object structure at runtime (cut-out)

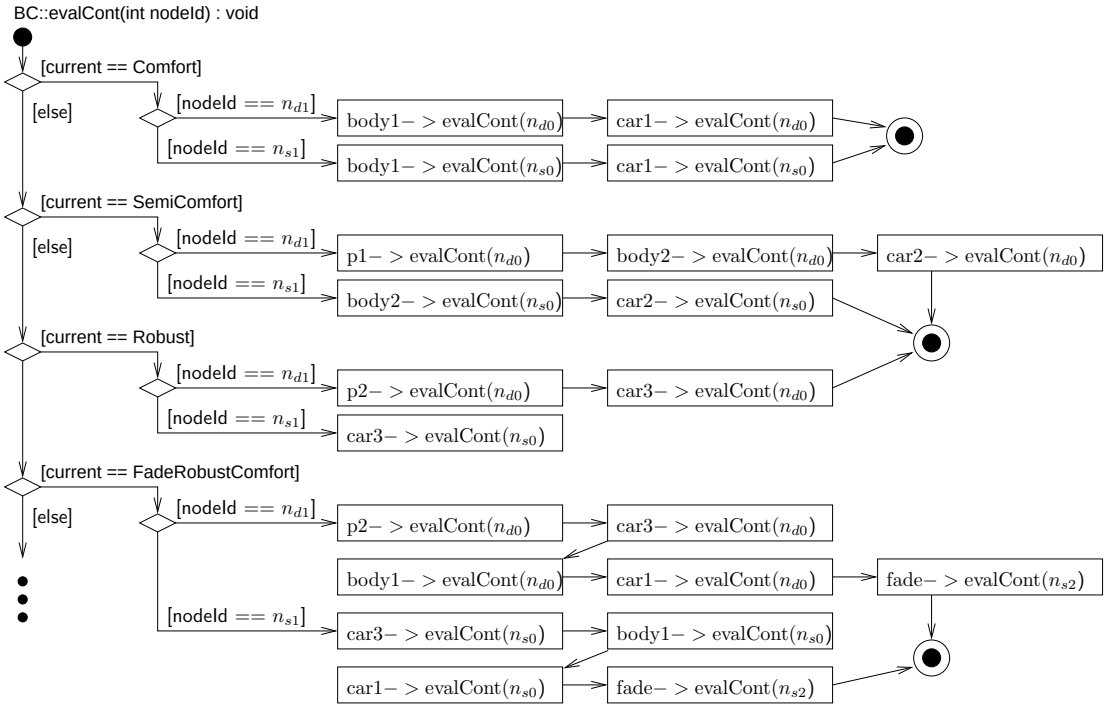


Figure 25: Activity Diagram of the evalCont(int nodeId) method of BC

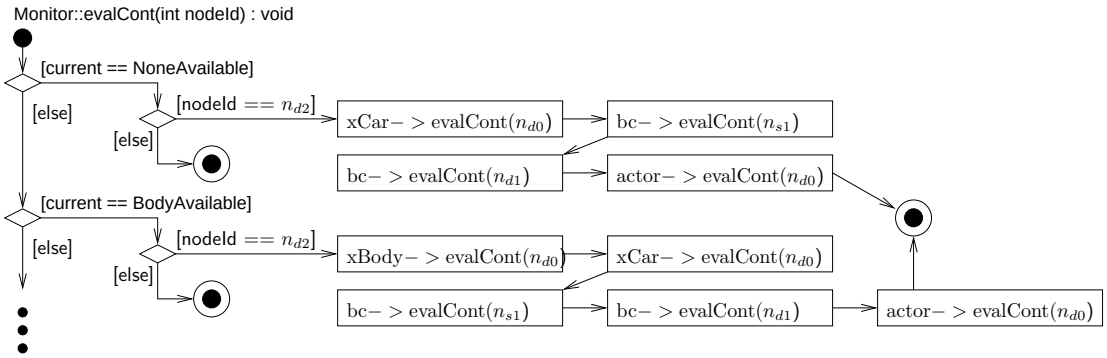


Figure 26: Activity Diagram of the evalCont(int nodeId) method of Monitor

from P controller p1, block n_{d0} from PIDT_1 controller body2, and block n_{d0} from PIDT_1 controller car2. As the number of states and nodes is finite (also in a concrete situation during runtime), the list of

evaluation nodes is finite and therefore, the algorithm terminates.

Note that the evaluation nodes of the embedded components, which are executed when n_{d1} or n_{s1} is evaluated, are determined by the partitioning algorithm as shown in Figure 23. These nodes of the subordinated components are the nodes from their corresponding reduced evaluation graphs. Thus, no knowledge about the internal node structure of the embedded components is required inside BC.

The discrete evaluation node of BC is the node presented in Figure 22. As BC embeds only basic continuous components, no other discrete nodes have to be evaluated.

Figure 26 shows the implementation of the evalCont(int nodeId) method of the Monitor component in the form of an activity diagram. Due to a lack of space we do not show that a Monitor instance references the embedded sensor and actuator components and the BC component (similar to Figure 24). Because Monitor is a *self-contained* component, it consists of exactly one continuous evaluation node n_{d2}. This node is periodically evaluated at runtime.

4.4 Flexible Reconfigurable Hierarchical Hybrid Systems

For the flexible reconfigration behavior the earlier outlined dataflow integration can be used. At runtime the evaluation graphs are computed (as distinguished from the static approach in the last section). The evaluation graphs are traversed to determine the next possible node to evaluate. While this results in a rather bad performance, as long as only small parts of the system are executed under this regime the performance drawback can be more than compensated by the gained flexibility. The cause is that flexibility is the key enabler for self* properties (like self-adaption or optimization).

To also extend our modular evaluation scheme for flexible reconfiguration scenarios where we cannot encapsulate the externally relevant evaluation blocks upfront, we have to combine our scheme with the dataflow integration concept. For such a combination, besides the local execution logic, we have to distinguish, if the component (controller) is embedded into another reconfigurable component which is flexible or not.

(1) *Combine Flexible Reconfigurable Components*: If a flexible reconfigurable component is embedded by another flexible reconfigurable component, we simply have to arrange that the dataflow propagation also happens across the component boundaries.

(2) *Embedding a Component with interface Statechart*: In the case that a non-flexible reconfigurable component is embedded by a flexible reconfigurable component, we simply have to consider the evaluation blocks provided by the embedded component in its interface as nodes in the dataflow propagation. This results in an at runtime-computed evaluation order which respects what the embedded component offers. It is to be noted that the efficiency of the modular execution of the embedded component is not affected at all by the less efficient processing of the embedding component. Therefore, unless large parts of the system follow the flexible scheme we will not observe any relevant performance problems.

(3) *Embedded by a Non-Flexible Reconfigurable Component*: If we in contrast have the case that a non-flexible reconfigurable component embeds a flexible reconfigurable component, we have to face the problem that the non-flexible reconfigurable component requires information about the evaluation of the embedded component which cannot be provided offline. One option is to simply use a predefined interface for the flexible reconfigurable component which can then be used by the non-flexible reconfigurable component to follow the usual modular execution scheme. This, however, requires that all flexible reconfiguration steps within the flexible reconfigurable component respect the constraint present due to the interface. Therefore, after each configuration step this crucial constraint has to be checked at runtime. Alternatively, the non-flexible reconfigurable component can be switched to control the evaluation in the dataflow manner. However, the interface of the non-flexible reconfigurable component then requires that for the whole set of components employing together a dataflow evaluation scheme after each configuration step the constraint implied by the interface has to be checked at runtime.

5 Prevent Critical Runtime Failures

As outlined in more detail in [25], we can support the correct development of reconfigurable hybrid systems based on the introduced modular execution scheme. This support at first has to exclude acyclic global

evaluation graphs to ensure the proper modular execution of the continuous parts. In addition, the correct real-time behavior of the discrete reconfiguration steps has to be ensured. In addition, we will discuss means to also cover flexible reconfiguration using runtime checks.

5.1 Static Analysis for Reconfiguration

This subsection presents two ways to prove that the reconfiguration behavior, specified for a hierarchical parallel composed component, is a refinement of the non-composed component: First, we present a syntactic check which is applicable in many cases (see Appendix A.3 for a detailed formalization). If this check is not applicable, we apply model checking to prove the refinement.

Syntactic Checks Checking the refinement relation simply requires answering the following three questions:

1. Are the *implied state changes* possible, i.e. , does there exist a transition from the source state of the subordinated component to the target state?
2. Do not the temporal requirements of this transition contradict the temporal requirements of the superordinated component?
3. Can the transition of the subordinated component become triggered simply by raising events?

The answer of question three determines if the syntactic checks are applicable: If the transition of the subordinated component is simply associated with a signal and if the guard and the time guard are true, the reconfiguration is simply executed by raising the appropriate signal. Otherwise, a complex analysis is required to determine if raising the signal will lead to a correct reconfiguration in all cases. This analysis is done by model checking as presented below.

Obviously, the first question is answered positively if the third question is answered positively. In order to prove that the temporal specifications are not contradictory, the deadlines simply need to be regarded: The deadline interval d_{sub} of the subordinated component needs to be a subset of the deadline interval d_{sup} of the superordinated component: $d_{sub} \subseteq d_{sup}$.

Model checking The model checking has to prove if a parallel composition of the components may lead to undefined state combination. As we first apply the syntactic checks where possible, model checking just needs to be applied if complex analysis is required to obtain the information if a transition of a subordinated component can be triggered just by raising the corresponding signal.

As the interface state charts are regarded on this level of abstraction instead of the detailed behavior, specified by the corresponding hybrid reconfiguration chart, the complexity of this verification step is significantly reduced as the approach requires not do verify the complete system at all.

5.2 Runtime Checking for Flexible Reconfiguration

As presented in Section 4.4, we can differ between three scenarios for the flexible reconfiguration: (1) *Combine Flexible Reconfigurable Components*, (2) *Embedding a Component with an interface Statechart*, (3) *Embedded by a Non-Flexible Reconfigurable Component*. While case (1) and (2) are straight forward, the embedding of a flexible component in an inflexible solution is problematic. In this case the coverage of offline techniques such as static analysis are rather limited. If the flexible reconfiguration rules only result in a feasible finite number of reachable configurations, we can map the resulting state space on our known techniques which enumerate the configurations and employ the related offline analysis techniques. However, when the resulting number of configurations is too large or even unbounded, only runtime checking can be used.

Depending on the case of combination for the flexible and non-flexible reconfigurable components and size of the system which follows the dataflow scheme different schemes for the runtime checking become appropriate:

(1) If we have a small part which is executed using the dataflow scheme, it is feasible to check the required constraints when a flexible reconfiguration has occurred. Using a double buffering for the configuration management, the detection can then be used to rollback the configuration and continue to execute the known well-formed configuration (see shadow configuration [1]). This scheme of course requires that the initial configuration is well-formed which can be checked offline.

(2) In the case of larger parts which are executed using the dataflow scheme, it is often not feasible to check the required constraints when a flexible reconfiguration has happened. Instead, we can use known techniques to detect the problem during dataflow executing (see Section 2). However, this solution obviously can result in a rather late detection where no compensation is possible without losing at least one continuous evaluation cycle. If loosing one cycle can be tolerated, a rollback to the former configuration can resolve the problem. However, when no such degraded performance can be tolerated, the application of the flexible dataflow scheme is simply not appropriate as we cannot exclude flexible reconfiguration steps leading to not well-formed continuous models.

We can summarize that the flexible reconfiguration scheme can only be used if either the real-time constraints are rather relaxed or when due to the small size of the configurations the runtime checks can be done before activating a configuration.

6 Tool Support

In order to outline the achieved tool integration we will first discuss its realization and then outline the resulting capabilities using the convoy building example. We will mainly focus on the hybrid aspects and not on the flexible reconfiguration as this shows well the complete approach from modeling via code generation to simulation. The flexible reconfiguration, as mentioned in Section 3, would especially help to support the flexible evolution of the system in cases where we cannot statically model all possible reconfigurations of the system upfront.

6.1 Tool Integration

The presented seamless approach is realized by the Fujaba Real-Time Tool Suite. We integrate the Fujaba Real-Time Tool Suite with the CAE Tool CAMeL-View and therefore, use the ability of CAMeL-View to generate code. As shown in Figure 27 both tools export hybrid components, which are integrated into a hierarchical model as an input for the binding tool. The export of the tools includes already the transformations of the model formalism to a code formalism, like C++. Afterward, the binding tool combines the inputs and therefore, combines both formalisms.

The core of the tool is our MECHATRONIC UML approach. The specific characteristic is that all above mentioned formalisms are combined by only one meta-model which becomes possible by a well-defined union of all formalisms.

In Figure 28 a cut-out of the MECHATRONIC UML meta-model is depicted. Both the time-continuous behavior and structure as well as the continuous control behavior and structure are integrated in one meta-model. This is reflected in the Figure by *ContinuousComponent* and *DiscreteComponent*. Furthermore, the hybrid class (HybridComponent and the port classes ContinuousPort and DiscretePort.

The MECHATRONIC UML approach applies model-driven development to develop software systems at a high level of abstraction to enable analysis approaches like model checking [15] as shown in Section 5. Therefore, ideally, we start with platform-independent models as shown in Section 3 to enable the compositional formal verification. Afterward, the platform-independent model must be enhanced with platform-specific information to enable code generation. The needed platform-specific information are based on a platform model, which specifies the platform-specific worst case execution times. Furthermore, we add platform-specific information like priorities, periods, and deadlines to the models. After specifying the platform-specific information we can generate code from the models by considering the modular execution approach as presented in Section 4. We therefore apply three different models of abstraction, like the Model-Driven Architecture (MDA), with conformable transformations.

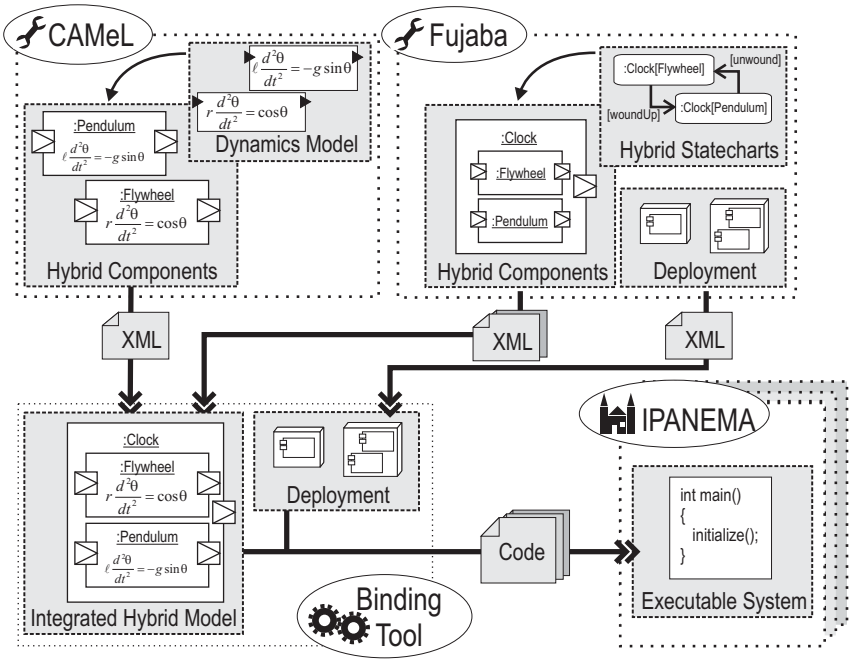


Figure 27: Tool Integration Overview [11]

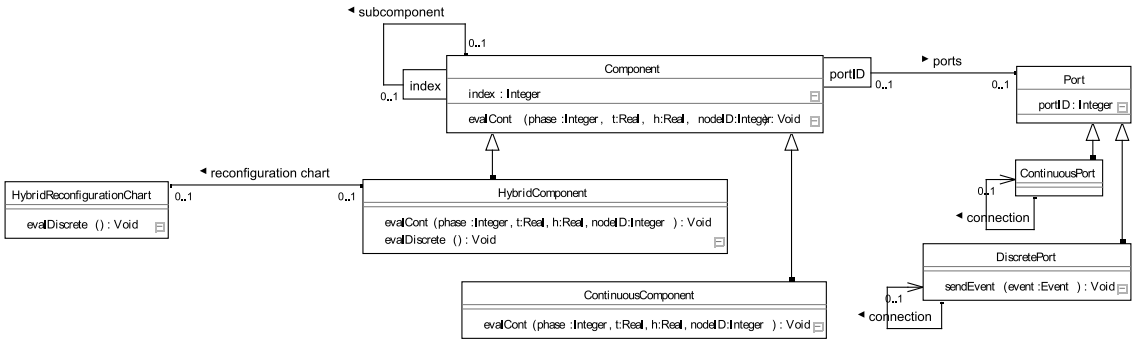


Figure 28: Cut-out of MECHATRONIC UML meta-model

6.2 Structural Modeling

We use the Fujaba Real-Time Tool Suite for modeling the architecture. Figure 29 and Figure 30 show a cutout of the internal structure of Shuttle. Similar to the Shuttle component, which is composed of multiple other component instances, the types of the subordinated instances are defined by further compositions. This leads to an architectural description of Shuttle, consisting of multiple layers.

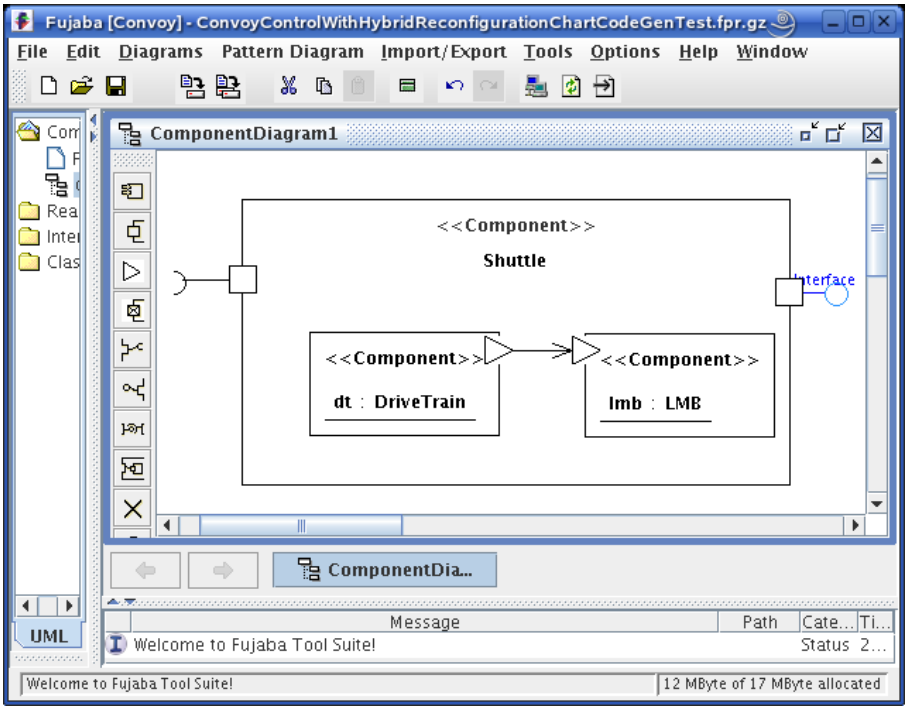


Figure 29: Structure of Shuttle

6.3 Behavioral Modeling

After modeling the structure of the shuttle convoy example, we specify the behavior of each component. We first show the specified discrete real-time behavior, on the level of the real-time coordination patterns and the individual discrete components which are modeled by the Fujaba Real-Time Tool Suite. The Shuttle's statechart realizes the discrete real-time coordination and it embeds –besides others– an instance of the hybrid component DriveTrain. The coordination leads to state changes, indicating if the shuttle is part of the convoy or not and if it is the leading shuttle of the convoy. Each of these states is associated with a different configuration where the DriveTrain-instance is in different states. Therefore, a state change of shuttle implies a state change of DriveTrain which implies a state change of AccelerationControl. This models reconfiguration via multiple hierarchical levels.

Figure 31 shows a hybrid reconfiguration chart that describes the behavior of the AccelerationControl component. It consists of three discrete states: The states VelocityControl, PositionControl, or PositionControlWithPilotControl are associated with the continuous controller component configurations which have been sketched previously.

6.4 Simulation & Runtime

As a prerequisite for simulation we require the exclusion of deadlock caused by reconfiguration. In most cases the required refinement relation can be guaranteed following outlined static analysis (see Section 5).

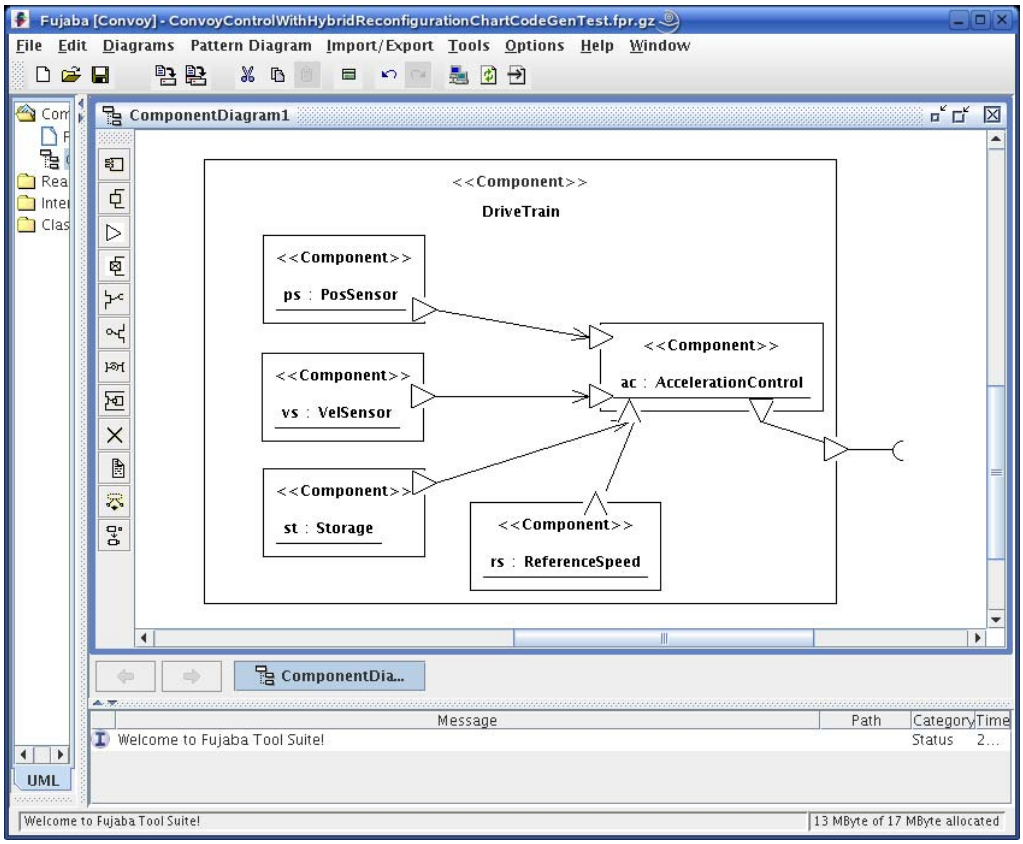


Figure 30: Structure of Drive Train

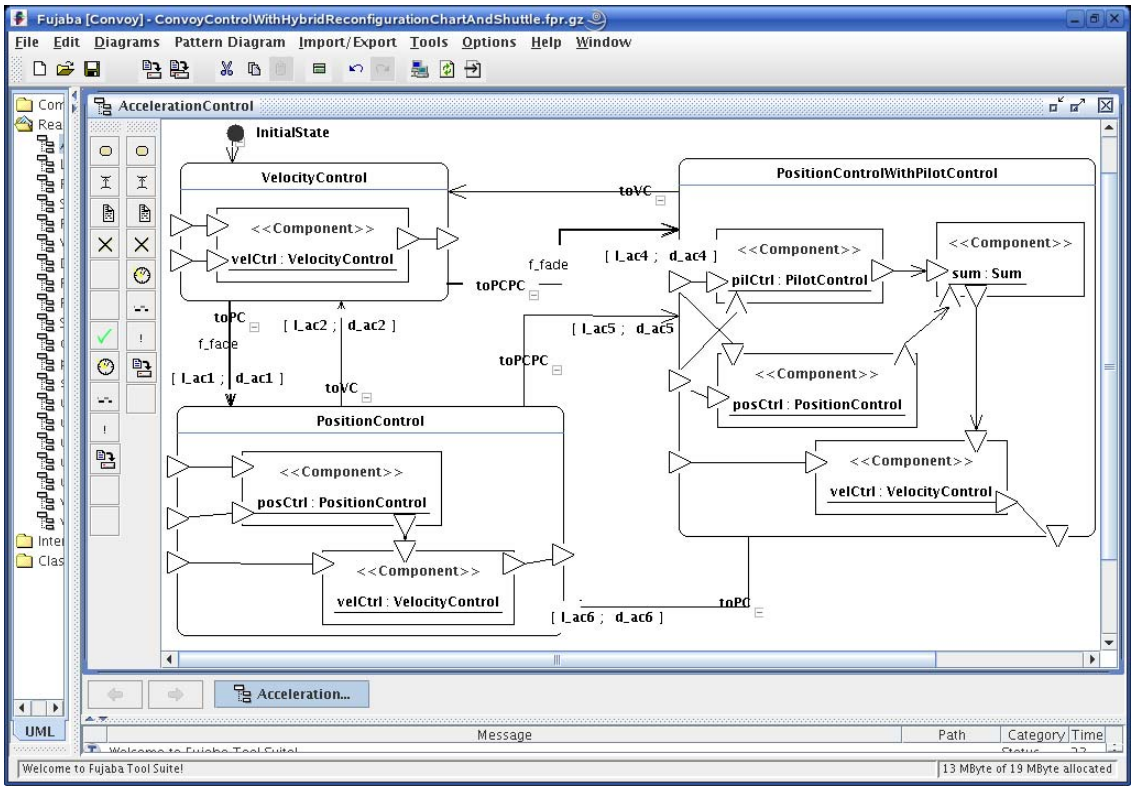


Figure 31: Hybrid reconfiguration chart of AccelerationControl with fading transitions

In several more complex cases, model checking for timed automata can be used (see [25, 26]) to check the refinement.[13]

After having checked the prerequisite, we generate C++ code from our model as described in Section 6.1 for a PC platform. After code generation and compilation, we show the behavior of the system using a simulation environment of the CAE (Computer Aided Engineering) Tool CAMeL-View. Figure 32 (simple convoy) shows two shuttles in a convoy (bottom-right window). On the upper left-hand plot the acceleration of the first and second shuttle is shown. On the upper right-hand plot the position of the shuttles are shown. On the bottom-left plot the reference values for the second shuttle is shown.

7 Conclusion and Future Work

The presented modular execution approach enables the modular execution for reconfigurable hybrid components. This is achieved by an expressive interface concept which ensures the proper handling of the embedded components using their reduced evaluation graph. The approach combines the advantage of the white-box scenario that works for any model without algebraic loops while avoiding its disadvantages such as monolithic execution logic and exponentially many evaluation orders in the case of large hybrid systems. Moreover, we present how even more flexible rule-based reconfiguration can be combined with the scheme.

We also presented how this modular execution scheme for reconfigurable hybrid systems could be further supported. We presented static analysis and verification techniques which guarantee that the reconfiguration itself does not result in undesired effects such as blocked or delayed reconfiguration steps. In addition, we present the tool integration between Fujaba and CAMeL-View which is based on the presented

[13] If there exists a complex interplay between the continuous and discrete behavior this would require the reachability analysis for the general form of hybrid systems which is undecidable [36]. Thus, we cannot expect to find an automatic solution for the general problem. However, the developed techniques cover most relevant cases for MECHATRONIC UML.

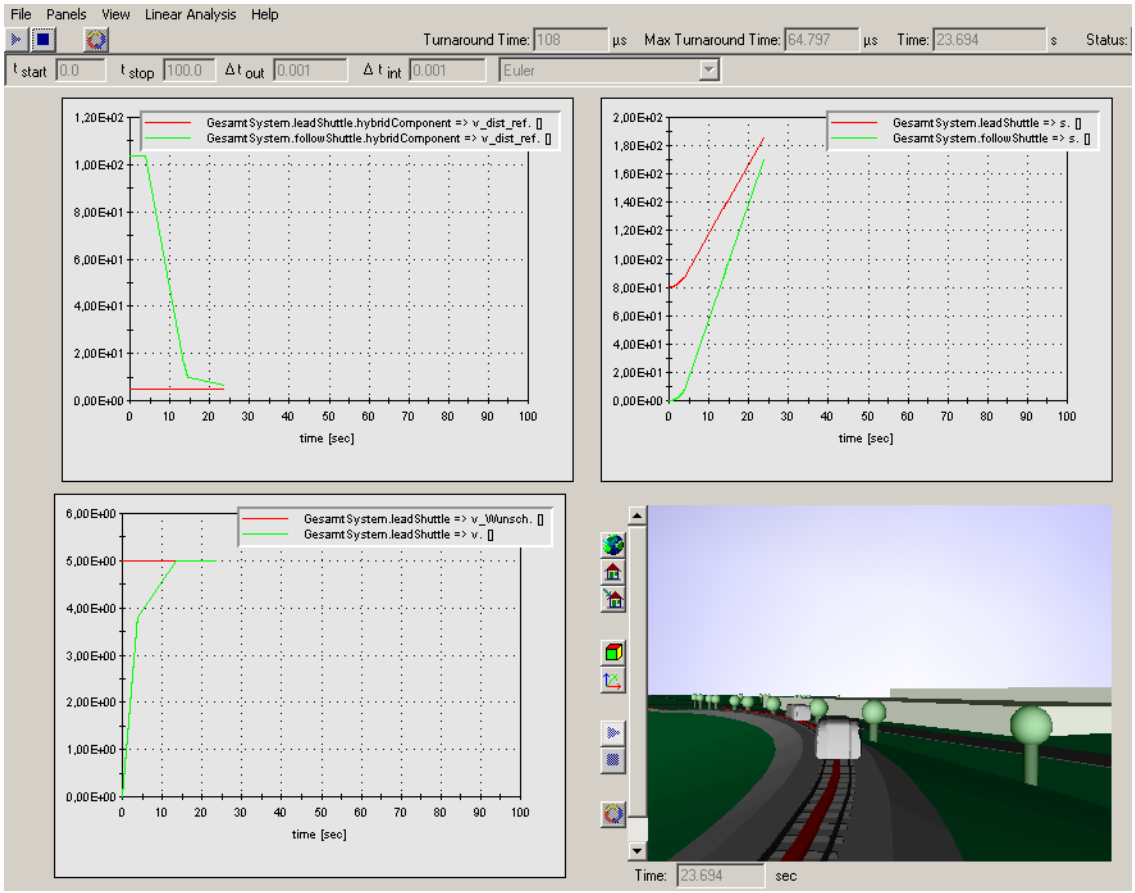


Figure 32: Simulation Environment

modular execution schemes effectively integrated models of both tools via a shared notion of a hybrid component with modular execution capabilities. We finally presented how the resulting reconfigurable hybrid models can be simulated and visualized within the CAMeL-View tool for validation purposes. This approach is currently evaluated by an industrial application (in a first step, the approach is implemented in an industrial tool).

The flexible methods for reconfiguration are very successful applied in first evaluations by students in form of master thesis and other student projects to the RailCab project (e. g. [31]). We present also in [34] an appropriate worst case execution approach for flexible reconfigurations. The formal verification of flexible reconfigurations is especially taken into account in [33, 37]. In the future, it is planned to further develop more support in particular for the flexible methods for reconfiguration. Also, the handling of the intertwined execution of the continuous and discrete parts can be further improved. The discrete statecharts often permit longer reaction times and require longer execution times than the continuous evaluation nodes. Therefore, the discrete parts do not have to be evaluated in every cycle and we might be able to optimize the overall performance by means of a clever schedule.

ACKNOWLEDGEMENTS

The authors thank all other PhD students and students who worked on the project. In particular we thank Sven Burmester for his substantial contributions and Tobias Eckardt for proof-reading. We further thank the reviewers for their detailed and helpful comments that helped to considerably improve the paper.

References

[1] Alimi, R., Y. Wang, and Y. R. Yang (2008). Shadow configuration as a network management primitive. In *SIGCOMM '08: Proceedings of the ACM SIGCOMM 2008 conference on Data communication, Seattle, WA, USA*, New York, NY, USA, pp. 111–122. ACM.

[2] Alur, R., T. Dang, J. M. Esposito, R. B. Fierro, Y. Hur, F. Ivancic, V. Kumar, I. Lee, P. Mishra, G. J. Pappas, and O. Sokolsky (2001). Hierarchical hybrid modeling of embedded systems. In *EMSOFT '01: Proceedings of the First International Workshop on Embedded Software*, London, UK, pp. 14–31. Springer-Verlag.

[3] Bender, K., M. Broy, I. Peter, A. Pretschner, and T. Stauner (2002, July). Model based development of hybrid systems. In *Modelling, Analysis, and Design of Hybrid Systems*, Volume 279 of *Lecture Notes on Control and Information Sciences*, pp. 37–52. Springer Verlag.

[4] Berkenkötter, K., S. Bisanz, U. Hannemann, and J. Peleska (2004). Executable hybrid uml and its application to train control systems. In H. Ehrig, W. Damm, J. Desel, M. Große-Rhode, W. Reif, E. Schnieder, and E. Westkämper (Eds.), *Integration of Software Specification Techniques for Applications in Engineering*, Volume 3147 of *Lecture Notes in Computer Science*, pp. 145–173. Springer Verlag.

[5] Berry, G. and L. Cosserat (1985). The esterel synchronous programming language and its mathematical semantics. In *Seminar on Concurrency, Carnegie-Mellon University*, London, UK, pp. 389–448. Springer-Verlag.

[6] Biernacki, D., J.-L. Colaço, G. Hamon, and M. Pouzet (2008). Clock-directed modular code generation for synchronous data-flow languages. *SIGPLAN Notices 43*(7), 121–130.

[7] Burmester, S. and H. Giese (2003, October). The fujaba real-time statechart plugin. In H. Giese and A. Zündorf (Eds.), *Proc. of the first International Fujaba Days 2003, Kassel, Germany*, Volume tr-ri-04-247 of *Technical Report*, pp. 1–8. University of Paderborn.

[8] Burmester, S., H. Giese, A. Gambuzza, and O. Oberschelp (2004, October). Partitioning and modular code synthesis for reconfigurable mechatronic software components. In C. Bobeanu (Ed.), *Proceedings of European Simulation and Modelling Conference (ESMc'2004), Paris, France*, pp. 66–73. EOROSIS Publications.

[9] Burmester, S., H. Giese, S. Henkler, M. Hirsch, M. Tichy, A. Gambuzza, E. Much, and H. Vöcking (2007, May). Tool support for developing advanced mechatronic systems: Integrating the fujaba real-time tool suite with camel-view. In *Proceedings of the 29th International Conference on Software Engineering (ICSE), Minneapolis, Minnesota, USA*, pp. 801–804. IEEE Computer Society Press.

[10] Burmester, S., H. Giese, and M. Hirsch (2005, October). Syntax and semantics of hybrid components. Technical Report tr-ri-05-264, Software Engineering Group, University of Paderborn, Germany.

[11] Burmester, S., H. Giese, and F. Klein (2004, September). Design and simulation of self-optimizing mechatronic systems with fujaba and CAMeL. In A. Schürr and A. Zündorf (Eds.), *Proceedings of the 2nd International Fujaba Days 2004, Darmstadt, Germany*, Volume tr-ri-04-253 of *Technical Report*, pp. 19–22. University of Paderborn.

[12] Burmester, S., H. Giese, E. Münch, O. Oberschelp, F. Klein, and P. Scheideler (2008). Tool support for the design of self-optimizing mechatronic multi-agent aystems. *Int. J. Softw. Tools Technol. Transf. 10*(3), 207–222.

[13] Burmester, S., H. Giese, and O. Oberschelp (2004a, January). Hybrid uml components for the correct design of self-optimizing mechatronic systems. Technical Report tr-ri-03-246, Software Engineering Group, University of Paderborn, Germany, Paderborn, Germany.

[14] Burmester, S., H. Giese, and O. Oberschelp (2004b, August). Hybrid uml components for the design of complex self-optimizing mechatronic systems. In H. Araujo, A. Vieira, J. Braz, B. Encarnacao, and M. Carvalho (Eds.), *Proc. of 1st International Conference on Informatics in Control, Automation and Robotics (ICINCO 2004), Setubal, Portugal*, pp. 222–229. INSTICC Press.

[15] Burmester, S., H. Giese, and W. Schäfer (2005, November). Model-driven architecture for hard real-time systems: From platform independent models to code. In *Proceedings of the European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'05), Nürnberg, Germany*, Volume 3748 of *Lecture Notes in Computer Science (LNCS)*, pp. 25–40. Springer Verlag.

[16] Burmester, S., H. Giese, and M. Tichy (2005, August). Model-driven development of reconfigurable mechatronic systems with mechatronic uml. In U. Assmann, A. Rensink, and M. Aksit (Eds.), *Model Driven Architecture: Foundations and Applications*, Volume 3599 of *Lecture Notes in Computer Science (LNCS)*, pp. 47–61. Springer Verlag.

[17] Caspi, P., D. Pilaud, N. Halbwachs, and J. A. Plaice (1987). Lustre: a declarative language for real-time programming. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, Munich, West Germany*, New York, NY, USA, pp. 178–188. ACM.

[18] Colaço, J.-L., A. Girault, G. Hamon, and M. Pouzet (2004). Towards a higher-order synchronous data-flow language. In *EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software, Pisa, Italy*, New York, NY, USA, pp. 230–239. ACM.

[19] Colaço, J.-L., B. Pagano, and M. Pouzet (2005). A conservative extension of synchronous data-flow with state machines. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software, Jersey City, NJ, USA*, New York, NY, USA, pp. 173–182. ACM.

[20] Delaval, G., A. Girault, and M. Pouzet (2008). A type system for the automatic distribution of higher-order synchronous dataflow programs. In *LCTES '08: Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems, Tucson, AZ, USA*, New York, NY, USA, pp. 101–110. ACM.

[21] Edwards, S. A. and E. A. Lee (2003). The semantics and execution of a synchronous block-diagram language. *Sci. Comput. Program. 48*(1), 21–42.

[22] Friesen, V., A. Nordwig, and M. Weber (1998). Object-oriented specification of hybrid systems using umlh and zimoo. In *Proceedings of the 11th International Conference of Z Users on The Z Formal Specification Notation, Berlin, Germany*, Volume 1493 of *Lecture Notes in Computer Science (LNCS)*, pp. 328–346. Springer Verlag.

[23] Giese, H., S. Burmester, W. Schäfer, and O. Oberschelp (2004, November). Modular design and verification of component-based mechatronic systems with online-reconfiguration. In *Proc. of 12th ACM SIGSOFT Foundations of Software Engineering 2004 (FSE 2004), Newport Beach, USA*, pp. 179–188. ACM Press.

[24] Giese, H., S. Henkler, M. Hirsch, V. Roubin, and M. Tichy (2008). Modeling techniques for software-intensive systems. In D. P. F. Tiako (Ed.), *Designing Software-Intensive Systems: Methods and Principles*, pp. 21–58. Langston University, OK.

[25] Giese, H. and M. Hirsch (2005, December). Checking and automatic abstraction for timed and hybrid refinement in mechtronic uml. Technical Report tr-ri-03-266, Software Engineering Group, University of Paderborn, Germany, Paderborn, Germany.

[26] Giese, H. and M. Hirsch (2006). Modular verification of safe online-reconfiguration for proactive components in mechatronic uml. In *Satellite Events at the MoDELS 2005 Conference*, Volume 3844 of *Lecture Notes in Computer Science*, pp. 67–78. Springer Berlin / Heidelberg.

[27] Giese, H., M. Tichy, S. Burmester, W. Schäfer, and S. Flake (2003, September). Towards the compositional verification of real-time uml designs. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE-11), Helsinki, Finland*, pp. 38–47. ACM Press.

[28] Grosu, R., T. Stauner, and M. Broy (1998). A modular visual model for hybrid systems. In *FTRTFT '98: Proceedings of the 5th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, London, UK, pp. 75–91. Springer-Verlag.

[29] Hahn, M. (1999). *OMD - Ein Objektmodell für den Mechatronikentwurf. Anwendung in der objektorientierten Modellbildung mechatronischer Systeme unter Verwendung von Mehrkörpersystemformalismen. Fortschritt-Berichte VDI. Reihe 20. Nr. 299*. Düsseldorf: VDI Verlag.

[30] Heimann, B., W. Gerth, and K. Popp (2001). *Mechatronik: Komponenten, Methoden, Beispiele*. München/Wien: Fachbuchverlag Leipzig im Carl Hanser Verlag.

[31] Henkler, S., M. Breit, C. Brink, M. Böger, C. Brenner, K. Bröker, U. Pohlmann, M. Richtermeier, J. Suck, O. Travkin, and C. Priesterjahn (2009, November). $frits^{Cab}$: Fujaba re-engineering tool suite for mechatronic systems. In P. V. Gorp (Ed.), *Proceedings of the 7th International Fujaba Days*, Eindhoven University of Technology, The Netherlands, pp. 25–29. accepted.

[32] Henkler, S. and M. Hirsch (2006, October). A multi-paradigm modeling approach for reconfigurable mechatronic systems. In *Proceedings of the International Workshop on Multi-Paradigm Modeling: Concepts and Tools (MPM06), Satellite Event of the the 9th International Conference on Model-Driven Engineering Languages and Systems MoDELS/UML2006, Genova, Italy*, Volume 2006/1 of *BME-DAAI Technical Report Series*, Budapest University of Technology and Economics, pp. 15–25.

[33] Henkler, S., M. Hirsch, C. Priesterjahn, and W. Schäfer (2010). Modeling and verifying dynamic communication structures based on graph transformations. In *Proc. of the Software Engineering 2010 Conference, Paderborn, Germany, 22.-26.2.2010*. accepted.

[34] Henkler, S., S. Oberthür, H. Giese, and A. Seibel (2010, 5). Model-driven runtime resource predictions for advanced mechatronic systems with dynamic data structures. In *In Proc. of 13th International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC)*, pp. 202–209. IEEE Computer Society Press.

[35] Henzinger, T. A., P.-H. Ho, and H. Wong-Toi (1995, December). Hytech: The next generation. In *Proceedings of the 16th IEEE Real-Time Symposium*, pp. 56–65. IEEE Computer Press.

[36] Henzinger, T. A., P. W. Kopke, A. Puri, and P. Varaiya (1998). What's decidable about hybrid automata? *Journal of Computer and System Sciences 57*, 94–124. A preliminary version appeared in the Proceedings of the 27th Annual Symposium on Theory of Computing (STOC), ACM Press, 1995, pp. 373-382.

[37] Hirsch, M., S. Henkler, and H. Giese (2008). Modeling collaborations with dynamic structural adaptation in mechatronic uml. In *SEAMS '08: Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, New York, NY, USA, pp. 33–40. ACM.

[38] Homburg, C. (1993). Entwurf und implementierung einer standardisierten beschreibungsform mechatronischer systeme. Master's thesis, University of Paderborn, Germany.

[39] Honekamp, U. (1998). *IPANEMA - Verteilte Echtzeit-Informationsverarbeitung in mechatronischen Systemen*. Ph. D. thesis, University of Paderborn, Germany.

[40] Kobryn, C. (2004, March). Expertr's voice: Uml 3.0 and the future of modeling. *Software and Systems Modeling 3*(1), 4 – 8.

[41] Köhler, H. J., U. A. Nickel, J. Niere, and A. Zündorf (2000). Integrating uml diagrams for production control systems. In *Proceedings of the* $22^{n.d}$ *International Conference on Software Engineering (ICSE), Limerick, Ireland*, pp. 241–251. ACM Press.

[42] Kratz, F., O. Sokolsky, G. J. Pappas, and I. Lee (2006). R-charon, a modeling language for reconfigurable hybrid systems. In *Hybrid Systems: Computation and Control*, Volume Volume 3927/2006, pp. 392–406. Springer Berlin / Heidelberg.

[43] Lee, E. A. and H. Zheng (2007). Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software, Salzburg, Austria*, New York, NY, USA, pp. 114–123. ACM.

[44] Liu, X., Y. Xiong, and E. A. Lee (2001, September). The ptolemy ii framework for visual languages. In *Proceedings of the IEEE 2001 Symposia on Human Centric Computing Languages and Environments (HCC'01), Stresa, Italy*, pp. 50–51.

[45] Lublinerman, R., C. Szegedy, and S. Tripakis (2009). Modular code generation from synchronous block diagrams: modularity vs. code size. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Savannah, GA, USA*, New York, NY, USA, pp. 78–89. ACM.

[46] Lublinerman, R. and S. Tripakis (2008). Modularity vs. reusability: code generation from synchronous block diagrams. In *DATE '08: Proceedings of the conference on Design, automation and test in Europe, Munich, Germany*, New York, NY, USA, pp. 1504–1509. ACM.

[47] Lynch, N., R. Segala, and F. Vaandrager (2001, Mar). Hybrid i/o automata revisited. In M. D. D. Benedetto and A. Sangiovanni-Vincentelli (Eds.), *In Proceedings of Hybrid Systems: Computation and Control, Fourth International Workshop (HSCC'01), Rome, Italy*, Volume 2034 of *Lecture Notes in Computer Science*, pp. 403–417. Springer Verlag. Long version is Technical Report MIT/LCS/TR-827, MIT Laboratory for Computer Science, July 2001.

[48] Malik, S. (1993). Analysis of cyclic combinational circuits. In *ICCAD '93: Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design, Santa Clara, California, United States*, Los Alamitos, CA, USA, pp. 618–625. IEEE Computer Society Press.

[49] Mosterman, P. (2002). Hybrsim - a modeling and simulation environment for hybrid bond graphs. *Journal of Systems and Control Engineering - Part I 216*(1), 35 – 46. LIDO-Berichtsjahr=2002,;.

[50] Mosterman, P. J. and G. Biswas (1995, May). Modeling discontinuous behavior with hybrid bond graphs. In *Proceedings of the Intl. Conference on Qualitative Reasoning, Amsterdam, the Netherlands*, pp. 139–147.

[51] Mosterman, P. J. and H. Vangheluwe (2004). Computer automated multi-paradigm modeling: An introduction. In *Journal on Simulation*, Volume 80, pp. 433–450. SAGE.

[52] Oberschelp, O., A. Gambuzza, S. Burmester, and H. Giese (2004a, July). Modular generation and simulation of mechatronic systems. In N. Callaos, W. Lesso, and B. Sanchez (Eds.), *Proceedings of the 8th World Multi-Conference on Systemics, Cybernetics and Informatics (SCI), Orlando, USA*, pp. 1–6. International Institute of Informatics and Systemics (IIIS).

[53] Oberschelp, O., A. Gambuzza, S. Burmester, and H. Giese (2004b, July). Modular generation and simulation of mechatronic systems. In N. Callaos, W. Lesso, and B. Sanchez (Eds.), *Proc. of the 8th World Multi-Conference on Systemics, Cybernetics and Informatics (SCI), Orlando, USA*, pp. 1–6. International Institute of Informatics and Systemics (IIIS).

[54] Oberschelp, O., C. Homburg, M. Deppe, A. Gambuzza, and J. Seuss (2001). Verarbeitungsorientierte darstellung verteilter hybrider systeme der mechatronik. In *5. Magdeburger Maschinenbautage*, Magdeburg, pp. 22–34. Otto-von-Guericke-Universität.

[55] Object Management Group (2003a). *UML 2.0 Infrastructure Specification*. Object Management Group. Document ptc/03-09-15.

[56] Object Management Group (2003b). *UML 2.0 Superstructure Specification*. Object Management Group. Document ptc/03-08-02.

[57] Object Management Group (2005, January). *Systems Modeling Language (SysML) Specification*. Object Management Group.

[58] RailCab (2004). The "rail technology" challenge. RailCab Project Homepage. http://nbp-www.upb.de/en/concept/challenge.php.

[59] Stauner, T. (2001). *Systematic Development of Hybrid Systems*. Ph. D. thesis, Technische Universität München, Germany.

[60] Stauner, T., A. Pretschner, and I. Péter (2001, October). Approaching a discrete-continuous uml: Tool support and formalization. In *Proceedings UML'2001 workshop on Practical UML-Based Rigorous Development Methods – Countering or Integrating the eXtremists*, Toronto, Canada, pp. 242–257.

[61] Zanella, M. C. and R. Stolpe (1999). Distributed hil simulation of mechatronic systems applied to an agricultural machine. In *DIPES '98: Proceedings of the IFIP WG10.3/WG10.5 international workshop on Distributed and parallel embedded systems*, Norwell, MA, USA, pp. 107–116. Kluwer Academic Publishers.

A Syntax and Semantics

A.1 Prerequisites

First, we have to define the employed basic mathematical notations:

We use $\mathbb{R}$ to denote the set of the real numbers, $\mathbb{N}_0$ to denote the natural numbers including 0, $[a, b]$ with $a, b \in A$ and $a \leq b$ to denote the interval of all elements $c \in A$ with $a \leq c \leq b$, $\wp(A)$ to denote the power set of A, and $[A \rightarrow B]$ and $[A \rightharpoonup B]$ to denote the set of total response partial functions from A to B. $EQ(V_l, V_r)$ denotes the set of all equations of the form $v_l = f^i(v_r^1, ..., v_r^n)$ with operations f^i of arity n and left- and right-hand side variables of the equation $v_l \in V_l$, $v_r^1, ..., v_r^n \in V_r$. $COND(V)$ denotes the set of all conditions over variables of V. The set of possible operations and constants is named *OP*.

As a special case we assume a set of operations $\{\perp_i\}$ which do not explicitly define for an equation $v_l = \perp_i(v_r^1, ..., v_r^n)$ any specific restrictions on the relation between the input and output trajectories. The set of all these operations is denoted by $OP_\perp$. The set of only fully deterministic input/output operations are denoted by OP_{det}.

Other than the vector equations usually used by control engineers, we use a set of variables V to denote each single value and describe the mapping by a function $[V \rightarrow \mathbb{R}]$. All values of a vector of the length n can be represented in a similar fashion as $[[0, n] \rightarrow \mathbb{R}]$.

$f \otimes g$ further denotes the composition of the two functions $f : A_1 \rightarrow B_1$ and $g : A_2 \rightarrow B_2$ with disjoint definition sets $A_1 \cap A_2 = \emptyset$ defined by $(f \otimes g)(x)$ equals $f(x)$ for $x \in A_1$ and $g(x)$ for $x \in A_2$. The combination of two updates $a_1 \oplus a_2$ further denotes the composition of the two functionals $a_1 : [A_1 \rightarrow B_1] \rightarrow [A_1' \rightarrow B_1']$ and $a_2 : [A_2 \rightarrow B_2] \rightarrow [A_2' \rightarrow B_2']$ with disjoint sets $A_1 \cap A_2 = \emptyset$ and $A_1' \cap A_2' = \emptyset$ defined by $(a_1 \oplus a_2)(x \otimes y) := a_1(x) \otimes a_2(y)$.

A *directed graph* is defined as a pair (N, E) with N a finite set of vertices and $E \subseteq N \times N$. We write $n \rightarrow n'$ if $(n, n') \in E$ and extend this to sequences of transitions such that $n \rightarrow_* n'$ if exists $k \geq 0$ and $n_0, \ldots n_k \in N$ such that $n_0 = n$, $n_k = n'$, and for all $0 \leq i \leq k-1$ holds $n_i \rightarrow n_{i+1}$. If $\rightarrow_*$ is irreflexive ($\nexists n with n \rightarrow_* n$) the directed graph is acyclic. For a graph $G = (N, E)$ we have the following additional defined terms: $d_{out}(n) := |\{n' \in N | (n, n') \in E\}|$ is the out-degree of node n, $d_{in}(n) := |\{n' \in N | (n', n) \in E\}|$ is the in-degree of node n, $N_{out} \subseteq N$ is the subset of output nodes with $\forall n \in N_{out}$ holds $d_{out}(n) = 0$, and $N_{in} \subseteq N$ is the subset of input nodes with $\forall n \in N_{out}$ holds $d_{in}(n) = 0$.

A.2 Integrated Modeling

In this section, we consider the syntax and semantics of the different modeling approaches for the control and software engineering domain and especially focus on an integrated approach supporting reconfigurable hybrid systems. Further, we will consider a flexible reconfiguration approach which is required due to the evolution of the system.

A.2.1 Continuous Models

Syntax A continuous block provides a sufficient syntactical structure for the used concept of differential equations. Its syntax is defined by Definition 1.

Definition 1 *A* continuous block M *is described by a 7-tuple* $(V^x, V^u, V^y, F, G, C, X^0)$ *with* V^x *the state variables,* V^u *the input variables, and* V^y *the output variables. For the implicitly defined state flow variables* $V^{\dot{x}}$ *and auxiliary variables* $V^a = V^y \cap V^u$*, the set of equations* $F \subseteq EQ(V^{\dot{x}} \cup V^a, V^x \cup V^u \cup V^a)$ *describes the flow of the state variables, the set of equations* $G \subseteq EQ(V^y \cup V^a, V^x \cup V^u \cup V^a)$ *determines the output variables, and* $X^0 \subseteq [V^x \rightarrow \mathbb{R}]$ *the set of initial states. The* invariant C *with* $C \in COND(V^x)$ *is further used to determine the set of valid states.*

A block M is only *well-formed* when for the system of differential equations $F \cup G$ holds that there are no cyclic dependencies, no double assignments, all undefined referenced variables are contained in $V^u - V^y$, and a value is assigned to all state variables ($V^{\dot{x}}$) and output variables (V^y).

We can compose two continuous models if their variable sets are not overlapping and the resulting sets of equations are well formed as follows:

Definition 2 *The* composition *of two continuous models* $M_1 = (V_1^x, V_1^u, V_1^y, F_1, G_1, C_1, X_1^0)$ *and* $M_2 = (V_2^x, V_2^u, V_2^y, F_2, G_2, C_2, X_2^0)$ *denoted by* $M_1 \| M_2$ *is again a continuous model* $M = (V^x, V^u, V^y, F, G, C, X^0)$ *with* $V^x := V_1^x \cup V_2^x$, $V^u := V_1^u \cup V_2^u$, $V^y := V_1^y \cup V_2^y$, $F := F_1 \cup F_2$, $G := G_1 \cup G_2$, C *is derived from* C_1 *and* C_2 *as* $C = \{(x_1 \otimes x_2) | x_1 \in C_1 \wedge x_2 \in C_2\}$*, and the set of initial states is* $X^0 = \{((l_1, l_2), (x_1 \otimes x_2)) | (l_1, x_1) \in X_1^0 \wedge (l_2, x_2) \in X_2^0\}$.

$M_1 \| M_2$ is well-formed iff $V_1^x \cap V_2^x = \emptyset$, $V_1^u \cap V_2^u = \emptyset$, $V_1^y \cap V_2^y = \emptyset$, and $F \cup G$ are well-formed. A composition is *consistent* if the resulting continuous model is well-formed.

Semantics The state space of a continuous behavior is $X = [V^x \to \mathbb{R}]$ which describes all possible assignments for the state variables. A trajectory $\rho_u : [0, \infty] \to [V^x \to \mathbb{R}]$ for the set of differential equations F and input $u : [0, \infty] \to [V^u \to \mathbb{R}]$ with $\rho_u(0) = x$ for the current continuous state $x \in X$ and $\rho_u(t) \in C$ for all $t \in [0, \infty]$ describes a valid behavior of the continuous system. The output variables V^y are determined by $\theta_u : [0, \infty] \to [V^y \to \mathbb{R}]$ using G analogously. The semantics for a continuous model M is given by all possible triples of environment and system trajectories (u, ρ_u, θ_u) denoted by $[\![M]\!]$.

A.2.2 Reconfigurable Hierarchical Hybrid Systems

Before defining the syntax and semantics of a hybrid reconfiguration automaton which can capture hybrid statecharts and interface statecharts, we define the syntax and semantics of a standard hybrid automaton with static interfaces similar to hybrid I/O automata [47].

Syntax The syntax of a hybrid automaton is defined by Definition 3:

Definition 3 *A* hybrid automaton *is described by a 6-tuple (L, D, I, O, T, S^0) with L a finite set of locations, D a function over L which assigns to each $l \in L$ a continuous model $D(l) = (V^x, V^u, V^y, F(l), G(l), C(l), X^0(l))$ (see Definition 1) with identical variable sets, I a finite set of input signals, O a finite set of output signals, T a finite set of transitions, and a set of initial states $S^0 \subseteq \{(l, x) | l \in L \wedge x \in X^0(l)\}$. For any transition $(l, g, g', a, l') \in T$ holds that $l \in L$ is the source-location, $g \in COND(V^x \cup V^u)$ the continuous guard, $g' \in \wp(I \cup O)$ the I/O-guard, $a \in [[V^x \to \mathbb{R}] \to [V^x \to \mathbb{R}]]$ the continuous update, and $l' \in L$ the target-location. For every $l \in L$ we require that $D(l)$ is well-formed.*

Definition 4 *The* interface *$I(M)$ of a hybrid automaton M is defined as the external visible event sets and input and output variables $(I - O, O - I, V^u - V^y, V^y - V^u)$.*

Note that the presented definition of a hybrid automaton easily permits to still employ the concepts of Real-Time Statecharts such as clocks. We simply have to define clock variables v_i whose values are determined by equations $\dot{v_i} = 1$ in F to encode this feature into a hybrid automaton.

The parallel composition of two hybrid automata is defined as follows:

Definition 5 *For two hybrid automata M_1 and M_2 the* parallel composition *($M_1 \| M_2$) results in a hybrid automaton $M = (L, D, I, O, T, S^0)$ with $L = L_1 \times L_2$, $D(l, l') = D_1(l) \| D_2(l')$, $I = I_1 \cup I_2$, $O = O_1 \cup O_2$. The resulting transition relation is $T = \{((l_1, l_2), g_1 \wedge g_2, g_1^i \cup g_2^i, (a_1 \oplus a_2), (l_1', l_2')) | (l_1, g_1, g_1^i, u_1, l_1') \in T_1 \wedge (l_2, g_2, g_2^i, u_2, l_2') \in T_2 \wedge g_1^i \cap (I_2 \cup O_2) = g_2^i \cap (I_1 \cup O_1)\} \cup \{((l_1, l_2), g_1, g_1^i, u_1, (l_1', l_2)) | (l_1, g_1, g_1^i, u_1, l_1') \in T_1 \wedge g_1^i \cap (I_2 \cup O_2) = \emptyset\} \cup \{((l_1, l_2), g_2, g_2^i, u_2, (l_1, l_2')) | (l_2, g_2, u_2, l_2') \in T_2 \wedge g_2^i \cap (I_1 \cup O_1) = \emptyset\}$. S^0 is defined as $S_1^0 \times S_2^0$.*

The automaton M is only well-defined when for all reachable $(l, l') \in L$ holds that $D((l, l'))$ is well-formed and the internal signal sets are disjoint ($(O_1 \cap I_1) \cap (O_2 \cap I_2) = \emptyset$). The composition of hybrid automata is only *consistent* when the resulting automaton is well-defined.

If the set of equations is well-formed for every location, it can be similar to the case of a continuous block represented by a corresponding directed acyclic evaluation graph. In the case of composition of two hybrid automata, we can derive the resulting directed acyclic evaluation graphs for every combination of locations by simply combining the graphs of both locations at the connected inputs and outputs. If the composition is well-formed, the resulting combined graphs are all acyclic. It is to be noted that the number of considered evaluation graphs grows exponentially.

In order to abstract from signals, e. g. , signals that are exchanged between automata, we define the *hiding of signals* as in Definition 6 which is taken from [13].

Definition 6 *For a hybrid automaton $M = (L, D, I, O, T, S^0)$ the* hiding *of some signals $A \subseteq I \cup O$ denoted by $M \backslash_A$ is defined as the hybrid automaton $M' = (L, D, I', O', T', S^0)$ with $I' = I - A$, $O' = O - A$, and $T' = \{((l, g, g^i - A, u, l)) | (l, g, g^i, u, l)) \in T\}$.*

Semantics For $X = [V^x \rightharpoonup \mathbb{R}]$, the set of possible continuous state variable bindings, the inner state of a hybrid automaton can be described by a pair $(l, x) \in L \times X$. There are two possible ways of state modifications: Either by firing an instantaneous transition $t \in T$ changing the location as well as the state variables or by residing in the current location which consumes time and alters just the control variables.

When staying in state (l, x), firing an instantaneous transition $t = (l', g, g^i, a, l'')$ is done iff

- the transition's source location equals the current location: $l = l'$,
- the continuous guard is fulfilled: $g(x \otimes u) = \mathrm{true}$ for $u \in [V^u \to \mathbb{R}]$ the current input variable binding,
- the I/O-guard is true for the chosen input and output signal sets $i \subseteq I$ and $o \subseteq O$: $i \cup o = \dot{g}$, and
- the continuous update still fulfills the invariant of the target location $a(x) \in C(l')$.

The resulting state will be $(l'', a(x))$ and we note this firing by $(l, x) \to_{(i \cup o)} (l'', a(x))$.

Iff no instantaneous transition can fire, the hybrid automaton resides in the current location l for a non-negative and non-zero time delay $\delta > 0$. Let $\rho_u : [0, \delta] \to [V^x \to \mathbb{R}]$ be a trajectory for the differential equations $F(l)$ and the external input $u : [0, \delta] \to [V^u - V^y \to \mathbb{R}]$ with $\rho_u(0) = x$. The state for all $t \in [0, \delta]$ will be $(l, \rho_u(t))$. The output variables $V^y - V^u$ and internal variables $V^y \cap V^u$ are determined by $\theta_u : [0, \delta] \to [V^y \to \mathbb{R}]$ using $G(l)$ analogously. We additionally require that for all $t \in [0, \delta]$ holds that $\rho_u(t) \in C(l)$.

The trace semantics is thus given by all possible infinite execution sequences $(u_0, l_0, \rho^0_{u_0}, \theta^0_{u_0}, \delta_0) \rightarrow_{e_0} (u_1, l_1, \rho^1_{u_1}, \theta^1_{u_1}, \delta_1) \ldots$ denoted by $[\![M]\!]_t$ where all $(l_i, \rho^i_{u_i}(\delta_i)) \rightarrow_{e_i} (l_{i+1}, \rho^{i+1}_{u_{i+1}}(0))$ are valid instantaneous transition executions.

Other aspects of hybrid behavior, such as zeno behavior and the distinction between *urgent* and *non-urgent* transitions, are omitted here. A suitable formalization can be found, e.g., in [35].

Hybrid Reconfiguration Automata In the following, we describe the syntax and semantics of hybrid reconfiguration automata that supports the flexible reconfiguration across module boundaries is defined formally.

Syntax We define the syntax of hybrid reconfiguration charts as in Definition 7.

Definition 7 *A* hybrid reconfiguration automaton *is described by a 6-tuple (L, D, I, O, T, S^0) with L a finite set of locations, D a function over L which assigns to each $l \in L$ a continuous model, $D(l) = (V^x(l), V^u(l), V^y(l), F(l), G(l), C(l), X^0(l))$ conf. to Definition 1, I a finite set of input signals, O a finite set of output signals, T a finite set of transitions, and $S^0 \subseteq \{(l, x) | l \in L \wedge x \in X(l)\}$ the set of initial states. For any transition $(l, g, g^i, a, l') \in T$ holds that $l \in L$ is the source-location, $g \in COND(V^x(l) \cup V^u(l))$ the continuous guard, $g^i \in \wp(I \cup O)$ the I/O-guard, $a \in [[V^x(l) \rightarrow \mathbb{R}] \rightarrow [V^x(l') \rightarrow \mathbb{R}]]$ the continuous update, and $l' \in L$ the target-location. For every $l \in L$ we require that $D(l)$ is well-formed.*

The automaton additionally allows that each location has its own variable sets. We use V^x to denote the union of all $V^x(l)$. V^u and V^y are derived analogously. We further use $V^x(F(l))$ to denote the *state variable set*. All assigned output variables are analogously named *provided output variable set* ($V^y(F(l) \cup G(l))$) and all input variables used are named *required input variable set* ($V^u(F(l) \cup G(l))$).

Definition 8 *The* (static) interface $I(M)$ *of a hybrid reconfiguration automaton M is defined as the external visible event sets and input and output variables $(I - O, O - I, V^u - V^y, V^y - V^u)$.*

The parallel composition of two hybrid reconfiguration automata can be defined as follows:

Definition 9 *For two hybrid reconfiguration automata M_1 and M_2 the* parallel composition *($M_1 \| M_2$) results in a hybrid reconfiguration automaton $M = (L, D, I, O, T, S^0)$ with $L = L_1 \times L_2$, $D(l, l') = D_1(l) \| D_2(l')$, $I = I_1 \cup I_2$, $O = O_1 \cup O_2$. $T = \{((l_1, l_2), g, , g', (a_1 \oplus a_2), (l_1^{'}, l_2^{'})) | (l_1, g_1, g_1', u_1, l_1^{'}) \in T_1 \wedge (l_2, g_2, g_2', u_2, l_2^{'}) \in T_2\} \cup \{((l_1, l_2), g_1', u_1, (l_1^{'}, l_2)) | (l_1, g_1, u_1, l_1^{'}) \in T_1\} \cup \{((l_1, l_2), g_2', u_2, (l_1, l_2^{'})) | (l_2, g_2, u_2, l_2^{'}) \in T_2\}$ is the resulting transition relation where $g(x, u, i, o) = g_1(x, u, i, o) \wedge g_2(x, u, i, o)$, $g_1'(x, u, i, o) = g_1(x, u, i, o) \wedge o \cap I_2 = \emptyset \wedge i \cap O_2 = \emptyset$, and $g_2'(x, u, i, o) = g_2(x, u, i, o) \wedge o \cap I_1 = \emptyset \wedge i \cap O_1 = \emptyset$.*

The automaton M is only defined when for all reachable $(l, l') \in L$ holds that $D((l, l'))$ is defined and the internal signal sets are disjoint ($(O_1 \cap I_1) \cap (O_2 \cap I_2) = \emptyset$).

The parallel composition also follows directly from the non reconfigurable case. In the case of hybrid reconfiguration automata, a correct parallel composition has to ensure that for all reachable $(l, l') \in L$ holds that $D((l, l'))$ does not contain cyclic dependencies.

Further, we define the terms *fading location* and *regular location* in Definition 10 and *passive location* in Definition 11 similar to the ones presented in [13]. The fading locations are inspired by the idea of [47] and represent fading transitions as time consuming intermediate states.

Definition 10 *For a hybrid reconfiguration automaton $M = (L, D, I, O, T, S^0)$ a location $l_f \in L$ with $D(l_f) = (V^x(l_f), V^u(l_f), V^y(l_f), F(l_f), G(l_f), C(l_f), X^0(l_f))$ is a* fading location *iff*

- *the invariant consists of just one inequality with respect to the variable v and an upper bound d_{max}: $C(l_f) \equiv (v \leq d_{max})$,*
- *v is a clock: $\exists v \in V^x(l_f)$ with $(\dot{v} = 1) \in F(l_f)$,*
- *v is reset to zero when entering l_f: for all $(l, g, g', a, l_f) \in T$ holds that $(v = 0) \in a$,*
- *exactly one transition exists leaving l_f: $|\{(l_f, g, g', a, l') | (l_f, g, g', a, l') \in T\}| = 1$, and*
- *for this transition holds $g \equiv d_{min} \leq v \leq d_{max}$, $g' = true$, and $a = Id$.*

All non-fading locations are regular locations.

Definition 11 *For a hybrid reconfiguration automaton $M = (L, D, I, O, T, S^0)$ a regular location $l_p \in L$ is a* passive location *iff the location and all transitions leaving it have no continuous constraints.*

Like in the case of the hybrid automata hold that if the set of equations is well-formed for every location, we have a corresponding directed acyclic evaluation graph. In the case of composition of two hybrid reconfiguration automata, we can derive the resulting directed acyclic evaluation graphs for every combination of locations by simply combining the graphs of both locations at the connected location-specific inputs and outputs.

Semantics The semantics can be derived from the hybrid automata semantics (see Section A.2.2) by always taking into account the location dependent notion $V^x(l)$ instead of the location independent V^x.

A.2.3 Flexible Reconfigurable Hierarchical Hybrid Systems

Syntax While graph transformation systems permit to model complex discrete models, we have to also support continuous behavior and thus in addition consider the following extensions for *hybrid graph transformation systems* (HGTS): We assign types to all nodes and edges, provide node type specific attributes $a \in \mathcal{V}$, and node type specific continuous behavior for the attributes of that node type. The state of a HGTS therefore consist of a graph G as well as an assignment X which provides for each node n and related attribute a the current value as $X(n, a)$.

Graph patterns are accordingly extended such that they can also contain Boolean constraints over the node attributes. In addition, for the right-hand side graph pattern we also permit to employ updates which determine the new attribute values as a function of node attributes of the left-hand side.

By permitting differential equations that define how the future state of a variable of node n can depend on the variables of all his neighbors, the HGTS can model in addition to its structure every possible evaluation graph. A well-formed HGTS state therefore requires that the evaluation graph is acyclic.

Semantics Due to the additional continuous behavior, we have, like for hybrid automata, two steps for HGTS: (1) At first, the classical GTS step results in a discrete change of the graph which takes place in zero time. In addition, the application of a rule requires that the additional Boolean constraints are fulfilled and may also change some of the attribute values of the nodes denotes by related attribute updates in the right-hand side of the rule. (2) Secondly, a time consuming continuous step that results for a current state (G, X), a time step $\delta > 0$, and trajectories $\rho : [0, \delta] \to [\mathcal{V} \to \mathbb{R}]$ with $\rho(0)(a) = X(n, a)$ for each node $n \in G$ and attribute $a \in \mathcal{V}$ which conforms to the continuous behavior specification of the type of n results in a state (G, X_δ) for time δ with identical graph G and the continuous state X_δ defined by $X_\delta(n, a) := \rho(\delta)(a)$.

Using this concept, we can, for example, describe the velocity of a vehicle given as a node n using differential equations or real-time behavior by means of clock variables with a constant derivative 1.

Using a rule labeling *alphabet* $\mathcal{A}$ and a corresponding *labelling* for all rules $\alpha : \mathcal{R} \to \mathcal{A}$ we further write $G \xrightarrow{a} G'$ instead of $G \to_r G'$ if $\alpha(r) = a$ and $G \xrightarrow{\tau(\delta)} G'$ for a continuous step. $G \xrightarrow{w}_* G'$ is accordingly defined as the finite sequence of steps $G_i \xrightarrow{a_i} G_{i+1}$ with w the concatenation of the labels $a_1; \ldots; a_n$ with $\tau(\delta_1); \tau(\delta_2)$ is reduced to $\tau(\delta_1 + \delta_2)$.[14] If more appropriate we may also omit the labeling and write $G \rightarrowtail G'$ and $G \rightarrowtail_* G'$.

The *composition* $S \oplus T$ of two HGTS S and T is defined by simply joining the rule sets. We can in addition define the *parallel composition* $S \| T$ of two labeled HGTS S and T by synchronizing always two rules of both HGTSs with the same label. We further require that to build a parallel composition, the combined rules are always identical w.r.t. shared element types. For the parallel composition holds that S as well as T simulate $S \| T$ (see below).

A.3 Static Analysis for Reconfiguration

To exclude runtime failures in the case of reconfiguration, static analysis can be done using the refinement notion related to the interface statecharts and special checked that ensure a correct embedding.

A.3.1 Refinement and Abstraction

To study what a correct relation between the realization of a component and its interface automaton is, we write for a possible execution sequence of states and transitions of a hybrid automaton $M = (L, D, I, O, T, S^0)$ with $(u_0, l_0, \rho^0_{u_0}, \theta^0_{u_0}, \delta_0) \to_{e_0} (u_1, l_1, \rho^1_{u_1}, \theta^1_{u_1}, \delta_1) \in [\![M]\!]_t$ simply $(l_0, \rho^0_{u_0}(0)) \to_{(u_0, \rho^0_{u_0}, \theta^0_{u_0}, \delta_0)} (l_0, \rho^0_{u_0}(\delta_0)) \to_{e_0} (l_1, \rho^1_{u_1}(0)) \to_{(u_1, l_1, \rho^1_{u_1}, \theta^1_{u_1}, \delta_1)} (l_1, \rho^1_{u_1}(\delta_1))$ to represent the state changes in a more uniform manner. We thus have the concept of a hybrid path $\pi = (u_0, \theta^0_{u_0}, \delta_0); e_0; \ldots; (u_n, l_n, \theta^1_{u_n}, \delta_n); e_n$ such that we write $(l_0, \rho^0_{u_0}(0)) \to_\pi (l_n, \rho^n_{u_n}(\delta_n))$ iff it holds that $(l_0, \rho^0_{u_0}(0)) \to_{(u_0, \rho^0_{u_0}, \theta^0_{u_0}, \delta_0)} (l_0, \rho^0_{u_0}(\delta_0)) \to_{e_0} \ldots (l_n, \rho^n_{u_n}(0)) \to_{(u_n, l_n, \rho^n_{u_n}, \theta^n_{u_n}, \delta_n)} (l_n, \rho^n_{u_n}(\delta_n))$.

For $e'_i = e_i - (O \cap I)$ the externally relevant events and $\theta^i_{u_i} = \theta^i_{u_i}|_{V^y(l_i) - V^u(l_i)}$ the output minus the internal variables, we have an abstract path $\pi' = (u_0, \theta^0_{u_0}, \delta_0); e'_0; \ldots; (u_n, \theta^1_{u_n}, \delta_n); e_n; \ldots$ and we write $(l_0, \rho^0_{u_0}(0)) \Rightarrow_{\pi'} (l_n, \rho^n_{u_n}(\delta_n))$. Note that $w; e; w'$ with $e = \emptyset$ is collapsed to $w; w'$ as no externally relevant events are received or emitted. The offered discrete as well as continuous interactions for a state (l, x) are further denoted by the set offer$(M, (l, x))$ which is defined as $\{e | \exists (l, x) \Rightarrow_e (l', x)\} \cup \{(du/dt)(0) | \exists (l, x) \Rightarrow_{(u, \theta_u, \delta)} (l, x')\}$.

An appropriate notion of hybrid refinement for the interface is then defined as follows:

Definition 12 *For two hybrid reconfiguration automata M_I and M_R holds that M_R is a* refinement *of M_I denoted by $M_R \sqsubseteq M_I$ iff a relation $\Omega \subseteq (L_R \times X_R) \times (L_I \times X_I)$ exists, so that for every $c \in (L_R \times X_R)$ a $c'' \in (L_I \times X_I)$ exists such that $(c, c'') \in \Omega$ and for all $(c, c'') \in \Omega$ holds*

$$\forall c \Rightarrow_\pi c' \quad \exists c'' \Rightarrow_\pi c''' \quad : (c', c''') \in \Omega \qquad \textit{and} \tag{1}$$

[14] We ignore here the problem of Zeno behavior which might result from an infinite sequence of classical steps or an non-convergent sequence of time steps ($\sum \delta_i \not\to \infty$).

$$\mathit{offer}(M_R, c) \supseteq \mathit{offer}(M_I, c'') \qquad \text{and} \tag{2}$$

$$\forall((l_R, x_R), (l_I, x_I)) \in \Omega : D^e(D_R(l_R)) \subseteq D^e(D_L(l_I)). \tag{3}$$

We denote dependencies between input and output variables using $D(M) \subseteq V^u \times V^y$. The external visible dependencies $D^e(M)$ are accordingly defined as $D^e(M) := D(M) \cap ((V^u - V^y) \times (V^y - V^u))$. Note that these dependencies essentially capture what has been formalized with the reduced evaluation graphs. What is not required for the considerations here is the accumulation of multiple inputs and outputs into separate evaluation blocks. Here, we are only interested in correctness and do not require this information which is essential for an efficient modular evaluation scheme.

A.3.2 Correct Embedding

Due to the fact that hierarchical composition in contrast to the general parallel composition restricts a potential overlapping of locations, the last subsection introduced refinement can be checked in many cases with static analysis without consideration of the full state-space of the model. In these checks, fading transitions and their durations play an important role. Formalizing their semantics leads to *simple interface automata* [13].

Definition 13 *An interface automaton $M = (L, D, I, O, T, S^0)$ is* simple *if it contains only passive and fading locations and two fading locations are never directly connected.*

We then can define the set H of possibly reachable state combinations of a reconfigurable hybrid component and its embedded occurrences as follows:

Definition 14 *Let $C = (S, M, P, prop)$ be a reconfigurable hybrid component with $M = (L, D, I, O, T, S^0)$, $S(l) = (I(l), B(l), E(l), map(l))$, and $E(l) = \{(N_1, (M_1, I_1, P_1, prop_1), l_1), \ldots, (N_n, (M_n, I_n, P_n, prop_n), l_n)\}$. The set of* possibly reachable states *is $H = \{(l, (l_1, \ldots, l_n))|l \in L\}$. We call $M\|_H(M_1\| \ldots \|M_n) := behavior(C)\|(M_1\| \ldots \|M_n)$ the* hierarchical parallel composition *of M and $M_1\| \ldots \|M_n$.*

The following theorem describes a simple syntactical rule which is sufficient to prove for the restricted case sketched above that a hierarchical parallel product does not have any timing errors. The basic idea is that the timing interval of a hybrid reconfiguration chart's fading transitions has to conform with one of its embedded simple interface state charts.

Theorem 1 *(see [13]) For the hierarchical parallel composition $M_1\|_H M_2$ of two hybrid automata M_1 and M_2 holds $M_1\|_H M_2 \sqsubseteq M_1\backslash_{I_2 \cup O_2}$ if*

1. *$I(M_1\|_H M_2) = I(M_1\backslash_{I_2 \cup O_2})$,*
2. *all initial states are also contained in H: $(\{(l_1, l_2)|(l_1, x) \in S_1^0 \wedge (l_2, y) \in S_2^0\} \subseteq H)$,*
3. *M_2 is a simple interface state chart (see Definition 13), and*
4. *for all $(l_1, l_2) \in H$ and transition $t_1 = (l_1, g_1, g_1^i, a_1, l_1') \in T_1$ holds:*
   - *(a) if l_1' is not a fading location, then for all $t_2 = (l_2, g_2, g_2^i, u_2, l_2') \in T_2$ with $g_1^i \cap (I_2 \cup O_2) = g_2^i$ must hold:*
     - *i. $g_2 = \text{true}$,*
     - *ii. l_2' is a passive location (see Definition 11), and*
     - *iii. $(l_1', l_2') \in H$.*

     In addition at least one such transition in M_2 must exist.
   - *(b) if l_1' is a fading location we can conclude that exactly one transition $t_1' = (l_1', g_1', g_1^{i\prime}, a_1', l_1'') \in T_1$ with $g_1' \equiv d_{min}^1 \leq v \leq d_{max}^1$ and $g_1^{i'} = \emptyset$ exists (see Definition 10). For any $t_2 = (l_2, g_2, g_2^i, a_2, l_2') \in T_2$ with $g_1^i \cap (I_2 \cup O_2) = g_2^i$ must hold:*

i. $g_2 = \mathrm{true}$,

ii. l'_2 *is a fading location, and*

iii. $(l'_1, l'_2) \in H$.

For the uniquely determined successor transition $t'_2 = (l'_2, g'_2, g_2^{i'}, a'_2, l''_2) \in T_2$ *with* $g'_2 \equiv d^2_{min} \leq v \leq d^2_{max}$ *must hold:*

iv. l''_2 *is a passive location (see Definition 11),*

v. $(l''_1, l''_2) \in H$*, and*

vi. $[d^2_{min}, d^2_{max}] \subseteq [d^1_{min}, d^1_{max}]$ *must be satisfied.*

Again, at least one such pair of transitions in M_2 *must exist.*

Theorem 1 can be extended to the general case of $M_S \|_H (M_1 \| \ldots \| M_n)$ by induction. Due to the syntactical check of Theorem 1, the hierarchical composition by means of the underlying hybrid control software cannot invalidate the timing properties ensured by the embedding hybrid statechart of the monitor (see [10] for details of the proof).