PARTITIONING AND MODULAR CODE SYNTHESIS FOR RECONFIGURABLE MECHATRONIC SOFTWARE COMPONENTS*

Sven Burmester† and Holger Giese
Software Engineering Group
University of Paderborn
Warburger Str. 100, D-33098 Paderborn, Germany
e-mail: [burmi|hg]@uni-paderborn.de

Alfonso Gambuzza and Oliver Oberschelp
Mechatronics Laboratory Paderborn
University of Paderborn
Pohlweg 98, D-33098 Paderborn, Germany
e-mail: [Alfonso.Gambuzza|Oliver.Oberschelp]@mlap.de

KEYWORDS
Hybrid Systems, Reconfiguration, Implementation.

ABSTRACT

Online reconfiguration offers great potential for improving the performance of technical systems at run-time. To enable reconfiguration, which can also be employed in complex hierarchical designs for mechatronic systems, support for the modeling and realization of modular reconfiguration is required. In this paper, we present the modular code synthesis for our modeling approach with Hybrid Components and a related Hybrid Statechart extension for the Unified Modelling Language (UML) which offers a representation of reconfigurable systems including conventional, non-reconfigurable block diagrams. To avoid the logic to be executed down to every possible ramification, the control behavior is at first partitioned into hierarchically organized continuous and discrete blocks. An implementation is then presented which schedules the evaluation of the continuous and discrete blocks in such a manner that the reconfiguration steps are safely and efficiently realized and the resulting code is suited to hard real-time systems.

INTRODUCTION

In mechatronic systems it is often necessary to adapt controllers to altered environmental conditions at runtime. Beside parameter adaptation it is in some cases necessary to alter the inner structure of the controller, for instance because of sensor failures. These structural changes have to be taken into account during modeling. If the structural variance is limited to predefined topologies in the first instance, the system is referred to as a reconfigurable one.

The inherent technical structure of the system can be used to define the reconfiguration. According to this approach a block-oriented model of a system consists of components describing a behavior (basic block) and of others describing a topology (hierarchy block). A hierarchy block describes relations and couplings between subordinated components while a basic block represents a directed input-output behavior in the shape of mathematical expressions which are specified by Ordinary Differential Equations (ODEs). The existing solvers for ODEs allow an application in hard real-time systems.

A reconfiguration is assumed to be an alteration of a hierarchy block. If possible configurations of a hierarchy are predefined, this is referred to as a statical reconfiguration. If possible configurations are assumed to be states of a hierarchy they can be represented as a statechart. The combination of statechart and configurable hierarchy is called a Hybrid Component.

As the reconfiguration is derived from the structure of the technical system, the principle of aggregation can be applied also to the information processing. In this case, each block makes up a module and the code can be generated separately. A hierarchy coordinates the exchange of data between modules and superior hierarchies.

A block diagram typically represents a forest of directed evaluation graphs by means of equations and couplings. If code is generated on the basis of this representation, the evaluation order of the equations of the system has to be calculated first. In the case of reconfigurable systems the evaluation order is usually not static, so a definite evaluation order is valid only for matching configurations. In combination with the module concept, code must be generated whose evaluation order can be controlled by the superior hierarchy.

The overall system makes up a so-called hybrid system. In the following, we will present the modular code synthesis for hybrid systems. In addition we will apply the modeling approach with Hybrid Components and a related Hybrid Statechart extension for the Unified Modelling Language (UML) which offers a representation of reconfigurable systems that includes classical, non-reconfigurable block diagrams [Giese et al., 2004, Burmester et al., 2004].

The paper is structured as follows: In the first section we introduce the testbed of a magnetic-levitation train and present the modeling of the controller switching. The

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

†Supported by the International Graduate School of Dynamic Intelligent Systems. University of Paderborn

following section will present our partitioning approach which is essential for modular code synthesis. We will then detail the implementation techniques and give an overview of the related work. A conclusion and an outlook to future work will round off the paper.

EXAMPLE AND MODELING

As an example we take the control system of a testbed of a magnetic-levitation train (Figure 1). The testbed allows only a control of the body mass in 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.

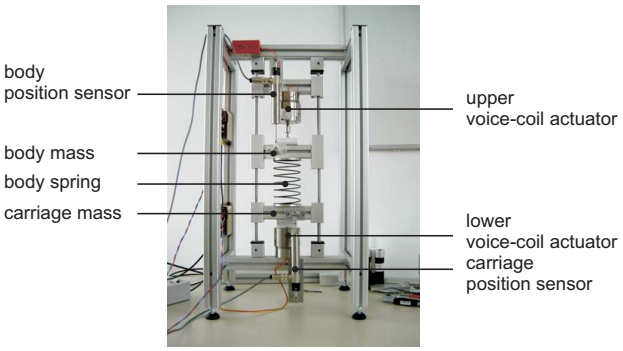

Figure 1: Testbed

Continuous Models

Multiple feedback controllers can be applied to the testbed. Figure 2 displays the three controllers that are applied. In Figure 2a the *Comfort* controller, providing the passengers the most comfort, is shown. It consists of two PIDT_1 controllers – one for controlling the undercarriage, the other for controlling the coach body. For inputs, $\text{PIDT}_{1,body}$ 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 $\text{PIDT}_{1,carriage}$. 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 2b. In case the $x^{body}_{current}$ sensor should fail, this controller structure could lead to an instability. Then the system needs to be reconfigured as shown in Figure 2c. The required position of the undercarriage is set to a constant value. This

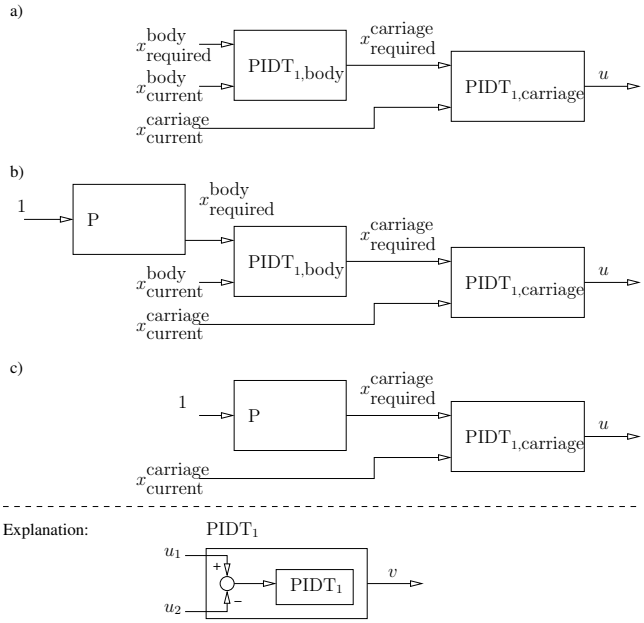

Figure 2: a) Comfort Controller, b) Semi-Comfort Controller, and c) Robust Controller

controller provides less comfort, but guarantees stability. To ensure stability, fault tolerance patterns are applied in such a way that the signal $x^{carriage}_{current}$ is never lost.

Hybrid Models

In order to model the exchange of single controller structures (reconfiguration) we use Hybrid Components and Hybrid Statecharts as described and formalized in [Giese et al., 2004, Burmester et al., 2004].

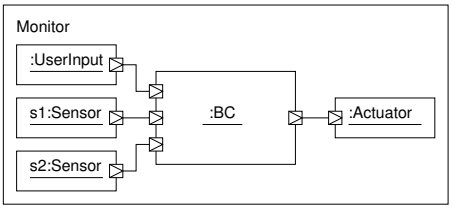

Figure 3: Structure of the controlling system

Figure 3 displays the structure of the controlling system. It consists of one Monitor component which embeds sensors, an actuator and a Body Control (BC) component; the latter contains the feedback controllers and switches between them.

The behavior of this hybrid BC component is specified by a simple Hybrid Statechart (see Figure 4) which is an extension of a Real-Time Statechart [Burmester and Giese, 2003]. It consists of 3 different control modes (discrete states) associated with the three configurations from Figure 2. The configurations consist of $\text{PIDT}_{1,carriage}$, $\text{PIDT}_{1,body}$, and the P blocks. Bold

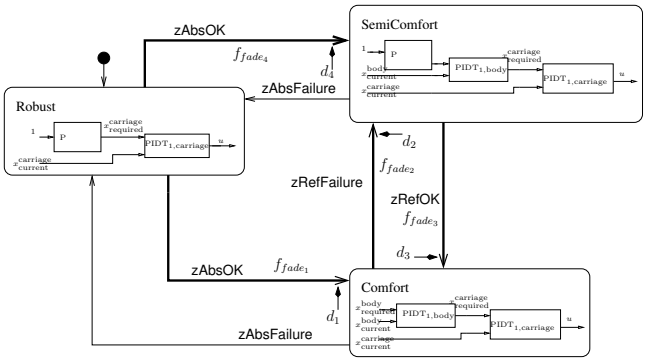

Figure 4: Behavior of the Body Control component

arrows indicate that output cross-fading 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. Thin arrows indicate that a switch is performed without time-consuming fading.

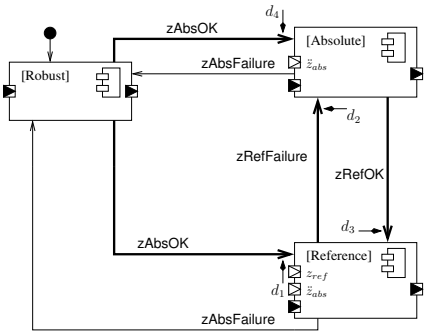

Figure 5: Interface Statechart of the Body Control component

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* of the component (see Figure 5). 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 [Oberschelp et al., 2004]). 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.

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 6) does not change at run-time although it consists of four discrete states. Therefore its Interface Statechart consists of just one state.

As displayed in Figure 3, the BC component is structurally embedded into the Monitor component. Figure 6 shows the behavior of the monitor embedding and coordinating the behavior of BC and the other embedded components as described in [Giese et al., 2004, Burmester et al., 2004]. For this embedding the notation from Interface Statechart (see Figure 5) is used.

The Hybrid Statechart in Figure 6 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 (iii + 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 e.g. 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.

PARTITIONING

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) and so forth.

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 single Hybrid Components.

Each Hybrid Component is partitioned into multiple discrete and continuous evaluation nodes [Oberschelp et al., 2004]. The continuous evaluation nodes compute continuous states and outputs of the feedback controllers, 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

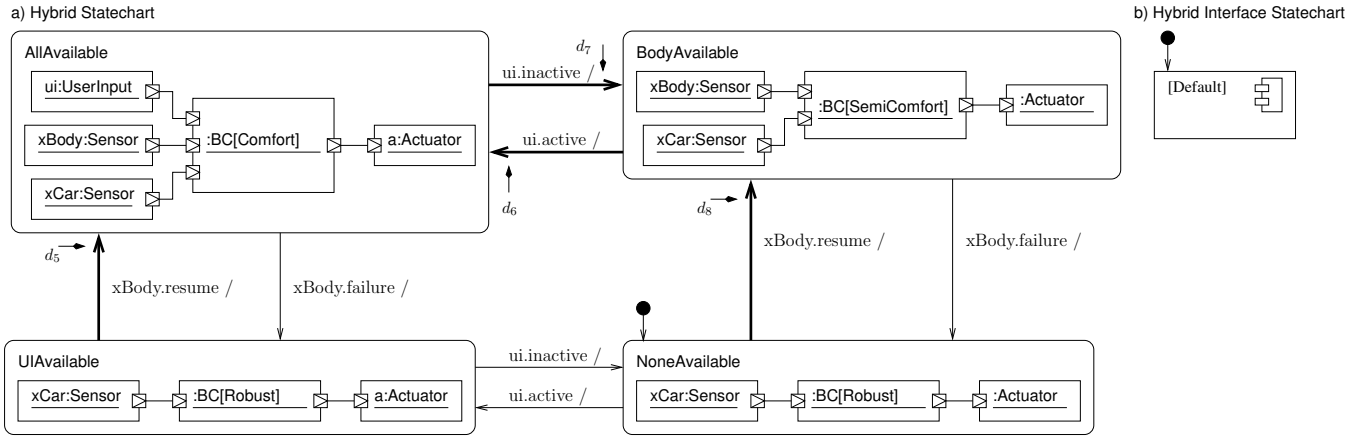

Figure 6: a) Behavior and b) Interface Statechart of the Monitor component

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

As already said, 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 from Appendix A which extends the partitioning algorithm from [Oberschelp et al., 2004] in order to build the superordinated evaluation nodes.

One of the nodes contains only expressions which influence only the inner continuous state of the controller. In [Oberschelp et al., 2004] this evaluation node is known as *state node (S)*. There is no particular evaluation order required for these expressions. Furthermore one evaluation node is created that determines outputs which do not depend on the current inputs (*non-direct link node (ND)*). Only the expressions whose outputs depend on the inputs (*direct link nodes (D)*) are partitioned into multiple evaluation nodes. 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 7a shows for example the P 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 7b. The P controller consists of one (direct link) node n_{d0}, the PIDT_1 controller consists of one (direct link) node n_{d0}

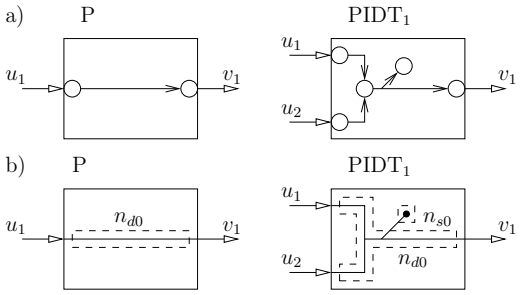

Figure 7: a) Evaluation node structure and b) *Reduced evaluation graph* of a P and a PIDT_1 controller

and a (state) node n_{s0}.

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.

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. Figure 8a shows the configuration of the SemiComfort state from Figure 4 (cf. Figure 2b). The partitioning results in the reduced evaluation graph of Figure 8b. Partitioning of the other configurations of Figure 4 (cf. Figures 2a and 2c) is done similar. Thus each of the three discrete states of the BC component is associated with another reduced evaluation graph.

As shown in Figure 6, the configuration that is associated with a discrete state does not have to consist exclu-

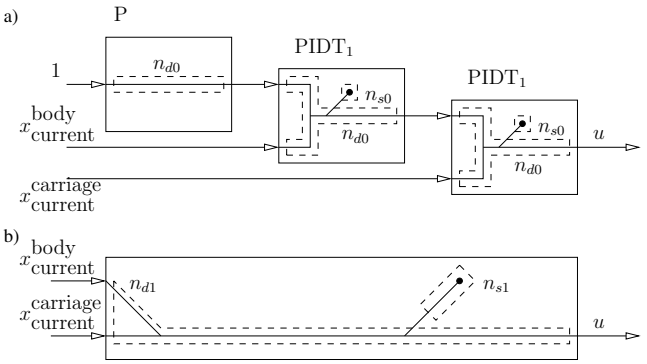

Figure 8: Configuration and reduced evaluation graph of the semi comfort controller

sively 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 one has 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.

IMPLEMENTATION

We have already stated that the evaluation order of the different nodes of a component depends on the external couplings. Thus an implementation must allow evaluation of the nodes in a different order, dependant on the couplings of the current configuration.

Basic Continuous Components

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 (cf. Figure 9). The possible nodeIds n_{d0} or n_{d0}, and n_{s0} resp. correlate to the nodes which have been determined by the partitioning algorithm (cf. Figure 7b), the internals (auxiliars[0]=...) are the expressions of the components (cf. Figure 7a).

Basic Discrete Components

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 exact one evaluation node. Furthermore it consists of an

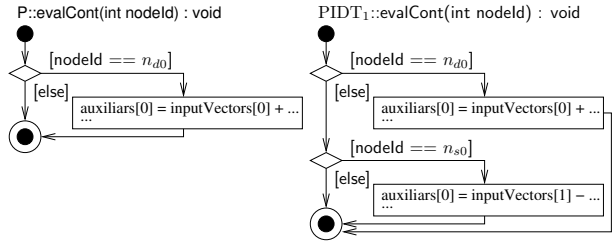

Figure 9: Activity Diagram of the evalCont(int nodeId) method of a P and a $PIDT_1$ controller

attribute current indicating the current discrete state.[1] 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 10.

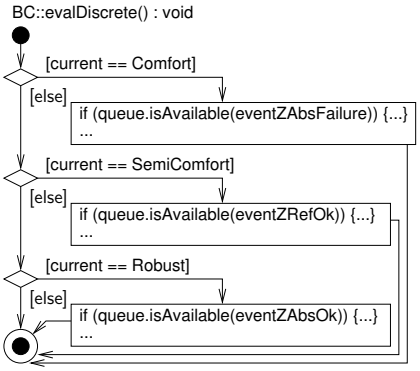

Figure 10: Activity Diagram of the evalDiscrete() method of BC

Hybrid Hierarchical Components

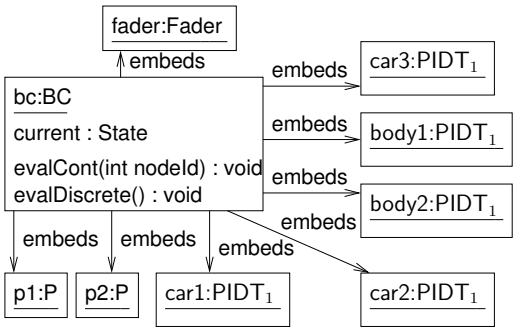

Figure 11: Object structure at run-time (cut-out)

An implementation of the hybrid hierarchical compo-

[1] In 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.

nents, 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 11 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 contains not expressions but lists of continuous nodes of the *embedded* (basic continuous or hybrid hierarchical) components. These lists are dependent on the current discrete state. Figure 12 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} from P controller p1, block n_{d0} from PIDT_1 controller body2, and block n_{d0} from PIDT_1 controller car2.
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 8. 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.
Furthermore there exists source code for the so-called *fading states* FadeRobustComfort, etc. These states belong to the underlying hybrid automata model, as presented in [Giese et al., 2004]. The component resides inside these states while fading transitions are executed. Note that in this case the nodes n_{d1} and n_{s1} are similar to a concatenation of the according nodes of states Robust and Comfort and an additional evaluation node of a Fader component. Due to limited space, not all fading states are displayed in Figure 12.
The discrete evaluation node of BC is the node presented in Figure 10. As BC embeds only basic continuous components, no other discrete nodes have to be evaluated.
Figure 13 shows the implementation of the evalCont(int nodeId) method of the Monitor component in 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 11). As Monitor is a *self-contained* component it consists of exactly one continuous evaluation node n_{d2}. This node is periodically evaluated at run-time.

RELATED WORK

Due to an ever-increasing complexity of technical systems methods for modular simulation have been developed in recent years. To be emphasized in particular are descriptor methods based on differential-algebraic equations (DAE). 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[2] or SIMPACK[3] which allow multi-body modeling and simulation yet have the same drawbacks in real-time evaluation as do all DAE-based tools [Hahn, 1999]. MATLAB/Simulink and Stateflow[4] are the de facto industry standard for the modeling of technical systems. Reconfiguration can be modeled by *conditionally executed subsystems* which are executed in dependency of a control signal. Although this approach allows modeling of reconfiguration, it has the disadvantage that systems will become very complex. The models consist of all active and inactive components. The use of our notion of Hybrid Statecharts results in multiple configurations, consisting of just the active components.
Hybrid bond graphs [Mosterman and Biswas, 1995] 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.
In [Mosterman, 2000] 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.

CONCLUSION AND FUTURE WORK

In this paper we presented a way to generate modular source code for reconfigurable, safety-critical systems. The reconfiguration was specified by our notion of Hybrid Components and Hybrid Statecharts.
The main problem to be solved in the implementation of feedback controllers and hybrid systems is the partitioning of the model into nodes and the determination of the

[2] www.adams.com
[3] www.simpack.com
[4] www.mathworks.com

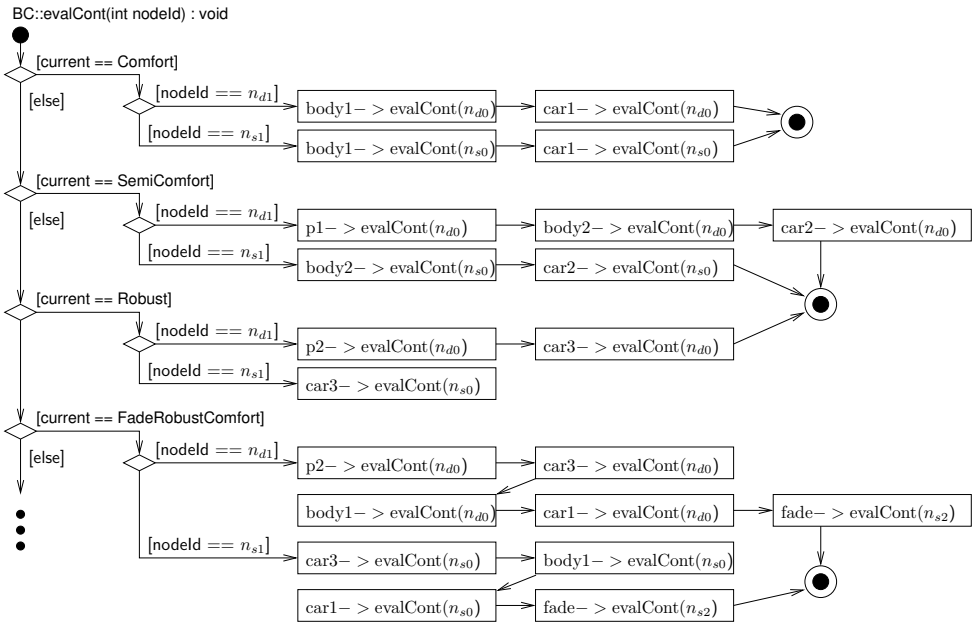

Figure 12: Activity Diagram of the evalCont(int nodeId) method of BC

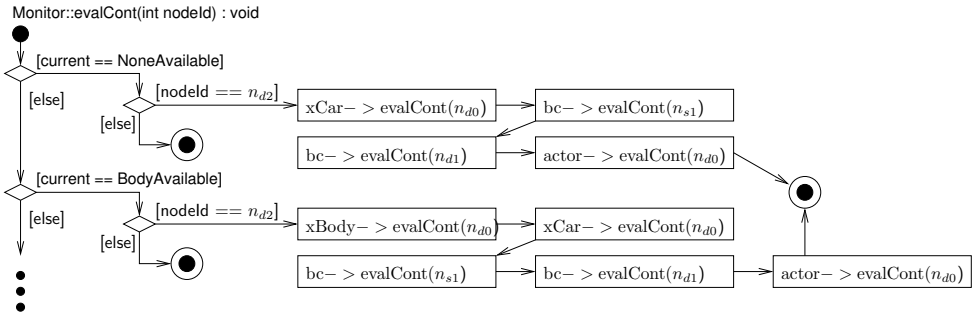

Figure 13: Activity Diagram of the evalCont(int nodeId) method of Monitor

node's evaluation order. In this paper we presented an advanced version of an algorithm that partitions basic continuous and discrete components, as well as hybrid hierarchical components. Then we showed how the partitioned model is mapped to modular structured code and presented a way of executing the code so as to ensure safe reconfiguration.

In current work we are implementing the automatic code generator and methods for the propagation of signals. Then we will test the presented example and evaluate the results.

ACKNOWLEDGEMENTS

The authors thank the students Vadim Boiko, Björn Schwerdtfeger, and Andreas Seibel for the interesting discussions.

REFERENCES

[Burmester and Giese, 2003] Burmester, S. and Giese, H. (2003). The Fujaba Real-Time Statechart PlugIn. In *Proc. of the Fujaba Days 2003, Kassel, Germany*.

[Burmester et al., 2004] Burmester, S., Giese, H., and Oberschelp, O. (2004). Hybrid UML Components for the Design of Complex Self-optimizing Mechatronic Systems. In *Proc. of 1st International Conference on Informatics in Control, Automation and Robotics (ICINCO 2004), Setubal, Portugal*. IEEE.

[Giese et al., 2004] Giese, H., Burmester, S., Schäfer, W., and Oberschelp, O. (2004). 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*. ACM. (accepted).

[Hahn, 1999] 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*. VDI Verlag, Düsseldorf.

[Mosterman, 2000] Mosterman, P. J. (2000). HYBRSIM - A Modeling and Simulation Environment for Hybrid Bond Graphs.

[Mosterman and Biswas, 1995] Mosterman, P. J. and Biswas, G. (1995). Modeling Discontinuous Behavior with Hybrid Bond Graphs. In *Proc. of the Intl. Conference on Qualitative Reasoning, Amsterdam, the Netherlands*, pages 139–147.

[Oberschelp et al., 2004] Oberschelp, O., Gambuzza, A., Burmester, S., and Giese, H. (2004). Modular Generation and Simulation of Mechatronic Systems. In *Proc. of the 8th World Multi-Conference on Systemics, Cybernetics and Informatics (SCI), Orlando, USA*.

APPENDIX A: PARTITIONING ALGORITHM

A single node is characterized by input, output, and state variables and a set of expressions with a left-hand-side variable and a right-hand-side expression with references to other variables. We derive an acyclic expression graph $G = (N, E)$ with node set N and edge set $E \subseteq N \times N$, where for each $n \in N$ and the related expression $v := \ldots v' \ldots$ holds that for each variable v' the expression refers to in the right-hand-side an edge $(n', n) \in E$ exists with n' the node related to variable v'.

For an acyclic graph $G = (N, E)$ we have the following additionally defined terms: $d_{out}((N, E), n) := |\{n' \in N | (n, n') \in E\}|$ the out-degree of node n, $d_{in}((N, E), n) := |\{n' \in N | (n', n) \in E\}|$ the in-degree of node n, $N_{out} \subseteq N$ the subset of output nodes with $\forall n \in N_{out}$ holds $d_{out}((N, e), n) = 0$, $N_{in} \subseteq N$ the subset of input nodes with $\forall n \in N_{in}$ holds $d_{in}((N, E), n) = 0$, and $N_{state} \subseteq N$ the subset of nodes which represent the internal state of the block.

The partitioning problem for a given acyclic expression graph $G = (N, E)$ of a block is how to determine a minimum number of partitions $N_1, \ldots, N_n \subseteq N$ such that: $N = N_1 \cup \ldots \cup N_n$ and $\forall i \neq j \; N_i \cap N_j = \emptyset$, 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 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 build by $N_1, \ldots, N_n$ and each node of $N' - N$ is also an acyclic graph (cf. [Oberschelp et al., 2004]). Such a minimal partitioning can be computed by means of the following algorithm:

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

  // 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
```