ABSTRACT

Simulation of complex mechatronic systems often requires decomposition of the model to be simulated. A decomposition aptly performed yields submodels that correspond to the technical structure of the modelled system. Thus a composition of models by means of existing components is made possible. Yet, coupling of the submodels affects the inner sequence of evaluation. This may result in deadlocks. We present an approach for the appropriate partitioning of the model code that allows keeping the inner evaluation flexible so that a deadlock-free coupling can be obtained. Submodels generated according to this approach have defined interfaces; therefore it is not necessary to consider the entire inner code for the sequence of evaluation to be known.

Keywords: Simulation, Mechatronics, Model Architecture, Code Generation

1. INTRODUCTION

The design of complex mechatronic systems has become so intricate that it can only be done by means of computer-aided modelling [1]. 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 shape fit for processing, their modular-hierarchical structure having to be observed [2].

A holistic, consistent, and structured concept for the design of mechatronic systems requires an exact preservation of the system structure from the model to the hard- and software on the processing level. If the design methodology is consistent, it allows a flexible adaption of the model to various scenarios, e.g., to a parallelized off-line simulation or distributed hardware-in-the-loop simulation (HILS), i.e., the direct allocation of information-processing components to physical aggregates used in mechanical engineering [3].

In the following, we will present in detail our concept for the modular generation of simulaable models of mechatronic systems on the basis of the inherent modular-hierarchical block structure of the technical system. We will also show our concept for the partitioning of modular model code. To exemplify this application, we employ the complex model of a testbed for a modern railway vehicle with an active suspension.

The paper is structured as follows: Section 2 presents the structuring of model code according to the technical structure. In Section 3 we describe different basic approaches to the decomposition of monolithic model code into modular one. We show our new approach for a modularized model code in Section 4. In Section 5 we depict the simulation environment, Section 6 concludes the paper.

2. STRUCTURING APPROACHES

In this section we discuss structuring approaches which can be regarded as basics for the decomposition of system models.

Modular-Hierarchical Structures

Modularization and hierarchical structuring are important aids for the solution of the complexity problem in technical systems. Modularizing the function yields function groups. In mechanical engineering this approach is realized as an aggregation. It allows aggregates to be designed, developed, tested, and manufactured independently of the entire system. Thus the overall functional problem is reduced to isolated sub-problems. Only in this way can products be developed and manufactured by different working teams and different suppliers.

According to this approach an entire system should be built up by hierarchically coupled software modules. In this scenario each module is generated independently and could in principle be executed on its own processor. But for this purpose we have to sever the dependencies between the modules. Dependencies are the couplings between the modules; the couplings in turn influence the internal evaluation order of each module. So we have to answer the question 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.

Block Diagrams

Hierarchical block diagrams are the usual method to describe technical systems. They are used in different domains, e.g., in mechanical and electrical engineering, informatics or systems
engineering. This common notation has its origin in control engineering. The blocks describe the behaviour of the system. They are structured by hierarchies describing the coupling between blocks and between hierarchies (Fig. 1). The behaviour is specified by mathematic equations or by means of a symbolic physical notation. Yet, the mathematical representation must be derived from the symbolic notation in order to make simulation possible:

The constraint saying that hierarchical elements must comprise only couplings can be used directly for a modularisation. Here, submodels are extracted and replaced by representatives that realize the coupling between the submodels:

3. MODULARISATION OF MODEL CODE

In this section we give an overview of common decomposition methods.

Execution Order of the Distributed System Model

For a correct execution of the simulation the overall evaluation order has to be taken into account in order to prevent a deadlock during computation of a simulation step that may block the entire simulation [4]. The connectibility 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 (Fig. 3) while in a white-box integration the communication routines are implemented purposefully into the evaluation functions of the subsystems. White-box integration solves the problems concerning the entire evaluation order but does not match our task to generate the modules independently (Fig. 6).

Black-Box Integration

In a black-box integration, 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 [4]:

![](image)

Fig. 1. Hierarchical block diagrams

Fig. 2 displays the decomposition approach mentioned above which enables us to assume that mechatronic systems consist of basic blocks (BB) and hierarchical blocks (HB) only.

![Diagram](image)

Fig. 2. Decomposition of block diagrams

In order to reduce dependencies, the equations are sorted according to the categories non-direct link, direct link, and state equations (Fig. 3). These categories serve to classify the input/output variables [5]:

- Non-direct links: All output variables that do not depend directly on input values are non-direct links and can thus be computed directly:

\[
\dot{y}_N(t) = f(x, p, t)
\]

(1)

- Direct links: This category comprises all input variables that have an immediate effect on at least one output variable:

\[
\dot{y}_D(t) = \dot{f}(x, y, p, t)
\]

(2)

- State variables: Input variables that apply only to state equations fall into this category:

\[
\dot{x} = \dot{f}(x, u, p, t)
\]

(3)

with \(\dot{x}\) the state vector, \(\dot{y}\) the output vector, \(u\) the input vector, \(p\) the parameter vector and \(t\) the time.

However, in principle, the problem of a communication deadlock cannot be fully solved just by introducing the ND-, D-, and S-blocks. A deadlock may occur between two subsystems if the couplings between their direct-link blocks make up a loop. The
model cannot be evaluated any more because the subsystems required to compute the data are waiting for the data of the other subsystems. This graph of waiting dependencies results in a loop and thus the subsystems block each other forever (Fig. 5).

**White-Box Integration**

Another solution is to use white-box communication. To this end, communications are required within the D code. A modular code generation is particularly difficult here since the sequence of the evaluation can be computed only for the entire system. This must be considered in the draft of the structure of the modular code:

Fig. 6. White-box integration

The proposed approach 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 [9] for the dynamic computation of the evaluation order at runtime).

**4. MODULAR CODE GENERATION**

This section presents the main interest of the paper. We will present a new decomposition procedure, called grey-box decomposition, and illustrate its useability by an example.

The actual example is an active vehicle suspension system with its controller which stems from the RailCab research project. The project was initiated and has been worked upon by several departments of the University of Paderborn and the Heinz Nixdorf Institute. In the project, a modular rail system will be developed; it is to combine modern chassis technology with the advantages of the Transrapid and the use of existing rail tracks [10]:

Fig. 7. Suspension module with controller

Fig. 7 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 can be damped actively by a displacement of their bases.

The control concept is based on a multi-level cascade control that results in hierarchically coupled controllers. For our purpose it is not necessary to go into the details of this control. The sequence of evaluation of the equations of the controller system changes with the respective state of the coupling. In order not to have to renew every component according to the chosen sequence of evaluation, a new procedure will be proposed.

With a grey-box method one tries to combine the advantages
of the white box with those of the black box. Determination of the sequence of evaluation here is done 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 saved. This implies an independent code generation 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 code of a module. This is especially important if one wants to forward code whose content is to be kept secret.

Let us now consider the first stage in more detail.

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. Appartenance of equations to a partial block is defined by their dependence on one or several input and output variables, with an input resp. an output variable being allocated definitely to just one partial block in the direct-link block.

The evaluation graph of a module corresponds to a forest consisting of different trees that cannot be decomposed further (Fig. 8).

In the case of an external coupling these subgraphs will possibly have to be sorted in order to comply with the sequence of evaluation. For an outward representation of the module it is sufficient to indicate the reduced graph, the interfaces, and the partitioned blocks. If the partitioned blocks are generated, e.g., as executable functions or objects, it will be possible to generate the entire module and/or compile it (Fig. 9). Thus it will be possible to generate modules and couple them subsequently as flexibly as by means of the white-box integration:

This procedure has the drawback of an overhead produced by the executable partitioned blocks. Also we will have to derive the sequence of evaluation of the entire module after the coupling and to control the evaluation at run-time.

**Partitioning Algorithm for Blocks**

In order to derive the sequence of evaluation we regard, the direct-link blocks. They are used to derive the so-called **acyclic expression graph**. A single block is characterized by input and output variables and a set of expressions with a left-hand-side variable and a right-hand-side expression with references to other variables. It can be represented by a corresponding directed acyclic expression graph when restricting the considered set of expressions to those where each variable occurs only once at the left-hand-side of an assignment expression. Each left-hand-side variable is represented by a node and all occurrences of variables in the right-hand-side as edge from the node of the referenced variable to that of the defined variable. Following this outline, we can 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 \( v := \ldots v' \) holds that for each variable \( v' \) the expression refers at in the right-hand-side an edge \( (n', n) \in E \) exists with \( n' \) the node related to variable \( v' \). When the graph \( G = (N, E) \) is embedded into a hierarchical element (cf. Fig. 2) the embedding graph \( G = (N, E) \) is called **context Graph** \( G' = (N', E') \). For an acyclic graph \( G = (N, E) \) we have the following additionally defined terms:

- \( d_{\text{out}}(n) := |\{n' \in N | (n, n') \in E\}| \) the out-degree of node \( n \).
- \( d_{\text{in}}(n) := |\{n' \in N | (n', n) \in E\}| \) the in-degree of node \( n \).
- \( N_{\text{out}} \subseteq N \) the subset of output nodes with \( \forall n \in N_{\text{out}} \) holds \( d_{\text{out}}(n) = 0 \).
- \( N_{\text{in}} \subseteq N \) the subset of input nodes with \( \forall n \in N_{\text{in}} \) holds \( d_{\text{in}}(n) = 0 \).

The partitioning problem for a given acyclic expression graph \( G = (N, E) \) with node set \( N \) and edge set \( E \subseteq N \times N \) of a block is how to determine a minimum number of partitions \( N_1, \ldots, N_n \subseteq N \) such that:

1. \( N = N_1 \cup \ldots \cup N_n \) and \( \forall i \neq j \ N_i \cap N_j = \emptyset \).
2. the derived graph \( G_p = (N, E_p) \) with \( N_p = \{N_1, \ldots, N_n\} \) and \( E_p = \{(N_i, N_j) | i \neq j \land i, j = 1, \ldots, n \land \exists n' \in N_i \land n'' \in N_j: (n', n'') \in E\} \) is acyclic, and
3. for any context graph \( G' = (N', E') \) with \( (N \cap N') \subseteq (N_{\text{in}} \cup N_{\text{out}}) \) 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.

In condition 1 we ensure that the partition sets \( N_i \) cover the full node set. The fact that the partitioning preserves the acyclic na-

---

**Fig. 9. Evaluation graph of a hierarchy**
ture is then enforced by condition 2. Finally, condition 3 informally states that the chosen partitioning, when embedded into an arbitrary context, only results in a cycle if the original graph also resulted in a cycle.

For a given acyclic graph \( G = (N, E) \) with node set \( N \) and edge set \( E \subseteq N \times N \), we can compute the required maximum partitioning of such a graph in two stages. Firstly, we compute the set \( D \) of all inputs that the node depends on in a forward topological traversal of the graph. Then we minimize this labelling by a backward traversal. The final partitioning is then defined to assign two nodes \( n, n' \) to the same partition \( (N_L[n]) \) iff \( L[n] = L[n'] \).

For \( E^* \) the transitive closure of \( E \), we can define the \( D(n) := \{ n'' \in N_n | (n, n'') \in E^* \} \) which to each node assigns all input nodes it depends on. The mapping finally computed \( L \), however, cannot that easily be defined in such an abstract manner. The resulting algorithm is presented in Fig. 10.

The algorithm requires roughly \( O(|N| + |E|) \) steps when ignoring the overhead for storing and manipulating the \( D \) and \( L \), because at each stage each node is visited via each incoming edge exactly once.

We thus have the following three cases to handle the block partitioning and execution order:

- **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 found 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 expressions within the code for those blocks.

- **Hierarchical Partitioning and Local Execution Order**: The block-partitioning algorithm outlined can be employed to derive also the required partitioning information in a hierarchical manner. The dependency graph which results from our embedding the block graphs of all embedded components instead of their expression graph is sufficient to derive again a minimum set of hierarchical blocks. 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 code generation in the white-box scenario.

If a block partitioning has been found, we also know that the original dependency graph 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.

\[ E^* := \bigcup_{i=0}^{\infty} E^i \] with \( E^0 = \{(n, n') | n \in N \} \) and \( E^{i+1} := \{(n, n'') | \exists n' \in N : (n, n') \in E^i \land (n', n'') \in E\} \)

\[ c : N \to N^i; \quad D : N \to \varphi(N_n); \quad L : N \to \varphi(N_n); \quad \text{related input nodes} \]

// phase 1:
// forward traversal to compute \( D \)
// \( D[n] \) all input nodes \( n \) depends on
// \( c[n] \) visited predecessor nodes
// for all \( n \) \( \in \) \( N \) do \( c[n] := \{ n \} \); done
// for all \( n \) \( \in \) \( N \) do \( D[n] := \{ n \} \); done

while ( \( F \neq \emptyset \) )
  for all \( n \) \( \in \) \( F \) do
    for all \( n' \) \( \in \) \( N \) with \( (n, n') \in E \) do
      \[ c[n'] := c[n'] + 1 \]
      if \( c[n'] = d_{in}(n') \) then
        \[ D[n'] := \bigcup (n'', n') \in E \) \( D[n''] \);
        \[ F := F \cup \{ n' \} \]
      fi
      done
    \[ F := F \setminus \{ n \} \];
done

// phase 2:
// backward traversal to compute \( L \)
// \( L[n] \) set of related input nodes
// \( c[n] \) visited successor nodes
// for all \( n \) \( \in \) \( N \) do \( c[n] := 0 \); done
// for all \( n \) \( \in \) \( N \) do \( D[n] := \{ n \} \); done

while ( \( F \neq \emptyset \) )
  for all \( n \) \( \in \) \( F \) do
    for all \( n' \) \( \in \) \( N \) with \( (n', n) \in E \) do
      \[ c[n'] := c[n'] + 1 \]
      if \( c[n'] = d_{out}(n') \) then
        \[ L[n'] := \bigcap (n', n'') \in E \) \( L[n''] \);
        \[ F := F \cup \{ n' \} \]
      fi
      done
    \[ F := F \setminus \{ n \} \];
done

// \( n \) and \( n' \) are in the same partitioning
// if \( L[n] = L[n'] \)
return \( L \);

Fig. 10. Block partitioning algorithm.
• **Global Execution Order:** When the top hierarchical level has been reached, the only remaining inputs and outputs are sensors resp. actuators. We can therefore be sure that the resulting partitioning graphs are acyclic and we can derive the main routine following the code generation scheme outlined for the hierarchical case.

To summarize, we can derive the code for each component in a modular manner by

• deriving the blocks, their code, and a block-dependency graph for all non-hierarchical components,
• deriving the blocks, their code, and a block-dependency graph for all hierarchical components considering only the internal coupling and the block-dependency graphs of all embedded components, and
• deriving the main routine code for the top-level component.

### 5. SIMULATION ENVIRONMENT

We model the technical system and continuous controllers by means of ordinary differential equations (ODE). A Computer-Aided Engineering (CAE) tool automatically derives differential equations for the mechatronic part which is described by multi-body system models [11]. For this purpose, nonlinear differential vector state equations are used as follows:

\[
\dot{x} = f(x, u, p, t) \\
y = g(x, u, p, t)
\]  

(4)

with \( \dot{x} \) the state vector, \( y \) the output vector, \( u \) the input vector, \( p \) the parameter vector and \( t \) the time. We use ODE solvers which are a part of the CAE environment for the computation of the model. This allows us to use the same model in offline as in online simulation (Hardware-in-the-loop).

### 6. CONCLUSION

Our grey-box approach results in modular code generation for the components such that the code generation for non-hierarchical components can be done entirely independently of that for hierarchical components. The code generation depends on the interfaces of the embedded components which include their block-dependency graphs. The approach includes the advantage of the white-box scenario that can be generated for any model without algebraic loop while avoiding the disadvantage of the white-box method that lies in the fact that the code generation is not modular and thus produces a bottleneck which brings about long turn-around times when simulating large mechatronic systems.

In the future, it is planned to develop methods for reconfiguration at run-time in order to make use of the full potential of the grey-box approach presented.

### 7. REFERENCES


