INTERNATIONAL DESIGN CONFERENCE - DESIGN 2012
Dubrovnik - Croatia, May 21 - 24, 2012.

MANAGEMENT OF CROSS-DOMAIN MODEL CONSISTENCY FOR BEHAVIORAL MODELS OF MECHATRONIC SYSTEMS

J. Rieke, R. Dorociak, O. Sudmann, J. Gausemeier and W. Schäfer

Keywords: consistency management, model transformation and synchronization, mechatronics, systems engineering

1. Introduction

Products of mechanical engineering and related industrial sectors, such as the automotive industry, are increasingly based on the close interaction of mechanics, electronics and software engineering, which is aptly expressed by the term mechatronics. Therefore, the development of such products is a highly interdisciplinary task. In order to support the development, a modeling language for the domain-spanning specification of the principle solution of an advanced mechatronic system has been developed in the Collaborative Research Center (CRC) 614. The principle solution is the result of the first, domain-spanning conceptual design phase and serves as a starting point for the following domain-specific concretization phase. In the course of the concretization, the involved domains work in parallel and use their own domain-specific methods, tools and models. As changes to these domain-specific models may affect other domains, the principle solution is further refined and extended during the concretization phase and becomes the domain-spanning system model.

Model transformation techniques are used to derive initial domain-specific models from the principle solution. Thus, we have a number of isolated, but interdependent models which capture the domain-specific aspects. The levels of abstraction differ between the models, i.e., domain-specific models contain more detailed information that should not be reflected in the domain-spanning system model.

The domain-specific models are then refined by the respective domains. Most changes only add more detailed information not relevant to other domains, but some changes may have an influence on the overall system model and other domains. One particular challenge is to ensure the consistency between the models during the whole development, as inconsistencies are likely to increase development time and costs. According to the methodology of the CRC 614, all domain-spanning relevant changes which occur in the involved domains during the concretization must be propagated to the domain-spanning system model. From the system model, those changes are propagated to the other domain-specific models. Changes in domain-specific models that are not considered to be domain-spanning relevant must not be propagated to the system model.

In general, model synchronization techniques can be employed in such a scenario. However, no techniques exist for model synchronization engines to distinguish domain-spanning relevant changes from domain-specific refinements. Thus, existing model synchronization approaches would propagate every change. Furthermore, even if we would be able to detect domain-specific refinements, we must take precautions not to lose such refinements on subsequent propagation of other changes.

As the novel contribution of this paper, we present a) a way to specify whether a change is a domain-specific refinement and thus not domain-spanning relevant, and b) an improved model synchronization technique which is capable of handling such refinements. We show how these techniques are employed to ensure the consistency of models on different abstraction levels.

We use the innovative railway system RailCab as a case study. RailCabs are small, driverless vehicles that autonomously drive on railway tracks and dynamically form convoys to save energy. In particular, we describe how discrete behavioral models are used for modeling the interaction between RailCabs; the focus lies on the formation of convoys. We show how such behavioral models are used in the domains software engineering and control engineering, which domain-specific refinements and domain-spanning relevant changes occur and how the latter are propagated.
For the software development we use MechatronicUML, a comprehensive technique for modeling and verifying the software of mechatronic systems with a special focus on real-time behavior, hybrid software components, and dynamic systems [Becker et al. 2011]. MechatronicUML is based on a subset of UML diagrams that supports the development of structural and behavioral aspects of mechatronic systems. For the development of the structure, a component-based approach is used. Components can communicate with each other by sending and receiving messages. Their behavior is specified by Real-Time Statecharts, which are based on UML state machines [Object Management Group 2010]. In the control engineering, the behavior of a system is specified using MATLAB/Simulink/Stateflow. In addition to the discrete behavior, the control engineer defines the RailCab's speed controllers.
The paper is structured as follows. In Section 2, we give an overview over the development process for mechatronic systems and the specification technique for the domain-spanning description of the principle solution. Furthermore, the running example is introduced. As the main contribution of this paper, we describe the challenges when synchronizing behavioral models during the development process and present our solution in Section 3. Finally, we conclude the paper in Section 4.

2. Development of advanced mechatronic systems

2.1 The design methodology for the domain-spanning conceptual design of mechatronic systems

Established design methodologies (cf. [Pahl et al. 2007] or the VDI Guideline 2206) focus on the particular disciplines; the system as a whole is only considered rudimentarily. This especially concerns mechatronic systems, as they are based on a close integration of mechanics, control engineering, electrics/electronics and software engineering. In the CRC 614, a new methodology for the domain-spanning conceptual design of mechatronic systems has been developed [Gausemeier et al. 2009a]. The core of the methodology is 1) the specification technique CONSENS (CONceptual design Specification technique for the ENgineering of complex Systems) and 2) a respective procedure model.

2.1.1 Procedure model for the development of advanced mechatronic systems

On the highest level of abstraction, the development process of mechatronic systems can be subdivided into the domain-spanning *conceptual design* and the domain-specific *concretization* (see Figure 1). Within the conceptual design, the basic structure, the operation modes of the system and its desired behavior are defined. The result of the conceptual design phase is the principle solution – a significant milestone in the mechatronic product development process. The domain-spanning specification of the principle solution forms the basis for the communication and cooperation of experts from different involved domains in the course of the further concretization.

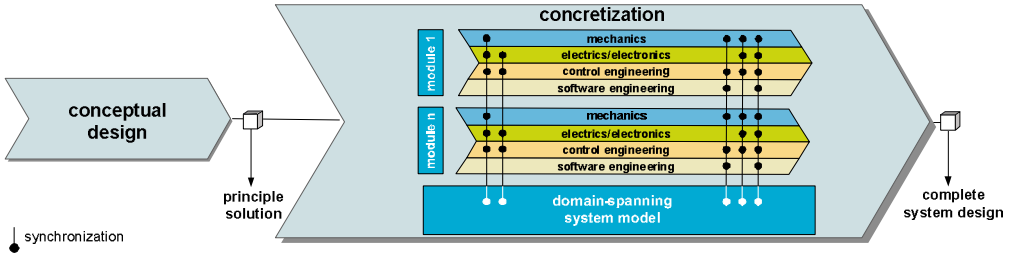

Figure 1. Synchronization of domain-specific models with each other and with the domain-spanning system model during the domain-specific concretization

During the concretization, the domains work in parallel using their domain-specific methods, tools and models (e.g., MechatronicUML for software engineering [Becker et al. 2011] and Simulink/Stateflow for control engineering). In the course of the concretization, the principle solution is further refined and extended and becomes the *domain-spanning system model*. Whenever a domain-spanning relevant change takes place, this system model has to be updated. For a more detailed description of the conceptual design, concretization and their particular phases, see [Gausemeier et al. 2009a].

2.1.2 Specification technique CONSENS

The specification technique CONSENS is used for the description of the principle solution of mechatronic systems. It is divided into different aspects (see Figure 2). These aspects are computer-internally represented as partial models. For a detailed description of the partial models, please refer to [Gausemeier et al. 2009a]. In contrast to other system modeling approaches such as UML [Object Management Group 2010] or SysML [Friedenthal et al. 2008], the specification technique is highly interconnected with the underlying procedure model and focuses strongly on mechatronic systems.

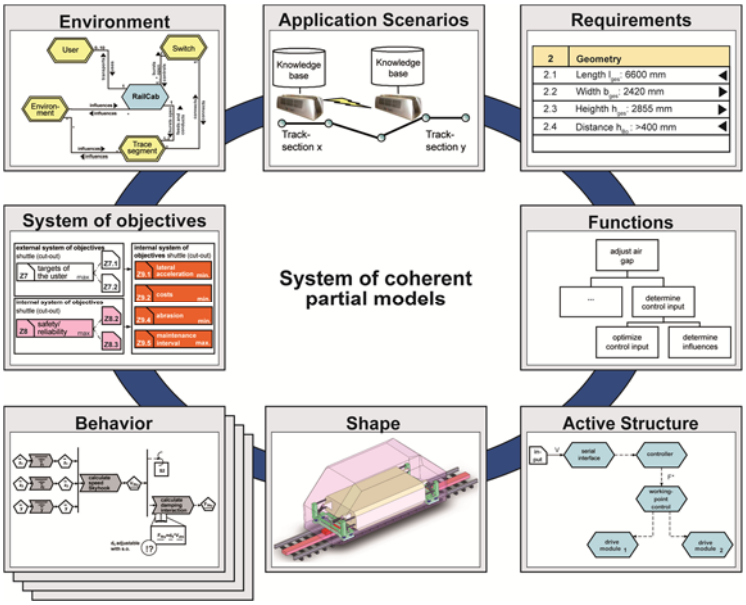

Figure 2. Aspects of the domain-spanning description of the principle solution

A dedicated software tool, the *Mechatronic Modeller*, has been developed [Gausemeier et al. 2010]; it supports this specification technique and the aligned design methodology. As the core of the tool, the metamodel for the specification technique defines which model elements are available during the description of the principle solution as well as how they are related to each other (abstract syntax). Furthermore, it defines how model elements have to be linked in order to have a meaning (static semantics) [Stahl et al. 2006]. A principle solution is computer-internally represented as a model, which is an instance of the metamodel. The use of a metamodel has a number of advantages, which were described in [Gausemeier et al. 2010]. In particular, a metamodel enables the use of automated transformation and consistency management techniques that are introduced Section 3.

2.2 Application example: the innovative railway system RailCab

The innovative railway system RailCab[1] is used as a demonstrator of the CRC 614 and serves as a case study throughout this paper. The core of the system consists of autonomous vehicles, called RailCabs, which transport passengers and goods according to individual demands rather than based on a timetable. To reduce the energy consumption, RailCabs may autonomously form convoys. A test facility on a scale of 1:2.5 has been built at the University of Paderborn (Figure 3 a).

[1] RailCab – Neue Bahntechnik Paderborn, project web site: http://railcab.de

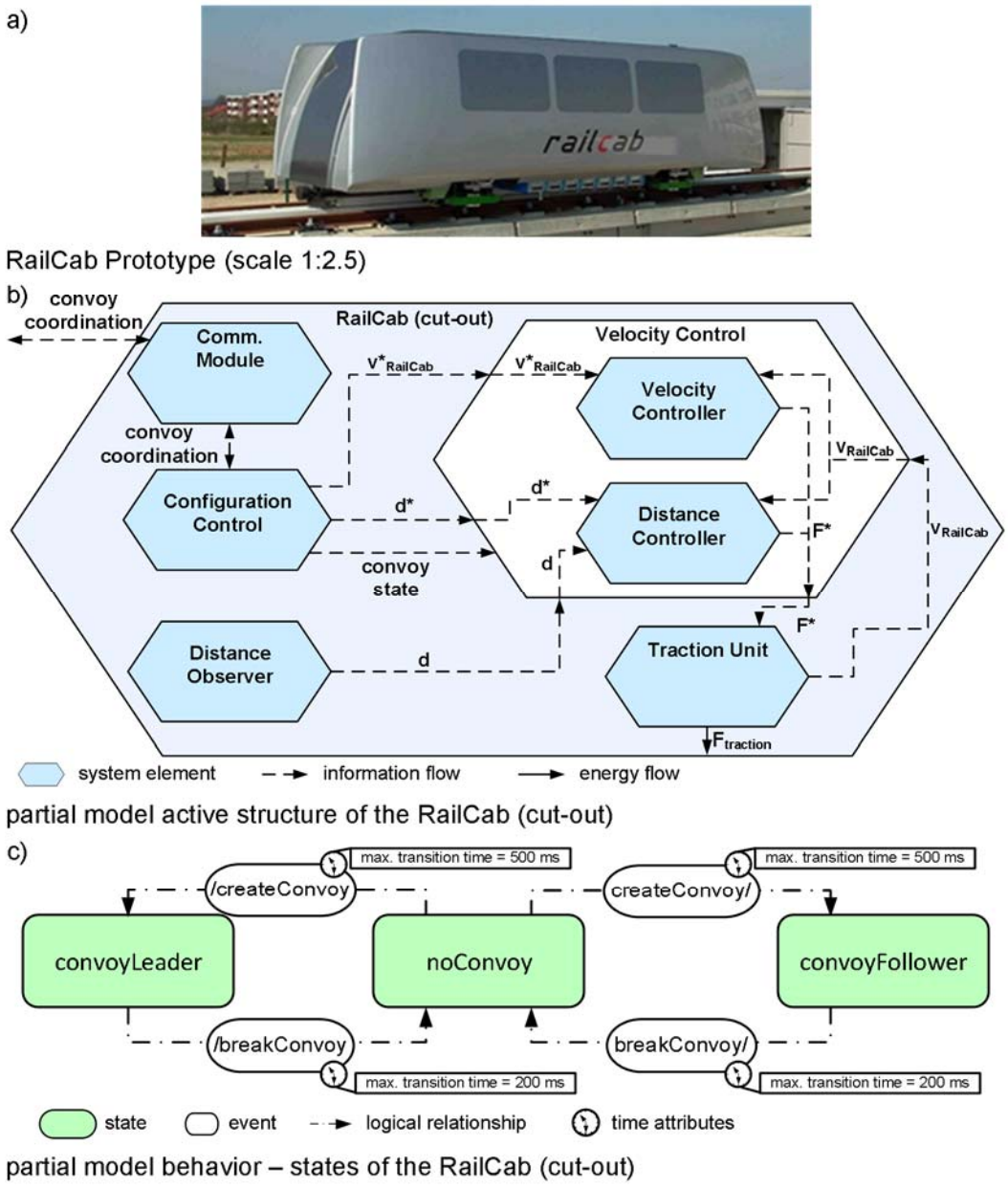

Figure 3. The innovative autonomous railway system RailCab

Figure 3 b) shows a cut-out of the active structure of the RailCab. In particular, it shows system elements responsible for the velocity control and the formation of convoys. Within a convoy, a RailCab can take the role of a leader or a follower. A follower RailCab must keep a safe distance to the RailCab ahead to prevent a collision. Two different control strategies for the speed of the RailCab are necessary. The default strategy is based upon sustaining a particular reference speed; it is implemented by the "Velocity Controller". The strategy for follower RailCabs in a convoy is based on keeping a particular safety distance to the RailCab ahead; it is implemented by the "Distance Controller". Thus, the "Distance Controller" is active as long as the RailCab has the follower role; otherwise, the "Velocity Controller" is active. The decision whether a convoy is formed or not is made by the system element "Configuration Control". In order to manage convoys, RailCabs must communicate with each other. The hardware level of this communication is realized via the system element "Communication Module". For further details on the RailCab's control strategy, please refer to [Henke et al. 2008].

During the domain-specific development, control engineers typically use Simulink block diagrams to refine the continuous behavior of the controllers. Additionally, the change of the controller strategies

must be specified in a discrete manner, e.g., using Stateflow. However, as the communication protocols for this discrete behavior are developed in the software engineering, it is crucial to synchronize the behavioral models of both domains. Initially, the Stateflow model and the behavior specification in the software engineering are based on the partial model behavior–states.
A cut-out of the partial model behavior–states that describes the formation of a convoy is shown in Figure 3 c). Three states are modeled: “noConvoy”, “convoyLeader”, and “convoyFollower”. At the beginning all RailCabs start in the state “noConvoy”. A RailCab can send a “createConvoy” message to the RailCab driving ahead. During the change to the state “convoyFollower”, the follower RailCab must adjust its speed with regard to the distance to the leader RailCab; the “Distance Controller” is thus activated. This change of the controller configuration and the exchange of messages cannot be carried out in zero time. Therefore, each transition is annotated with a maximum delay that the system needs for switching to the target state. For instance, it is specified, that the follower RailCab switches from the “noConvoy” to the “convoyFollower” state within 500 ms. While driving in a convoy, the follower RailCab can break the convoy by sending the “breakConvoy” message to the convoy leader.

2.3 Challenges for the consistency management in the domain-specific concretization

In this section, we illustrate the challenges for the consistency management of the different models with an exemplary development process. The starting point of the concretization is the domain-spanning principle solution. We map its model elements and relationships to the corresponding elements and relationships in the domain-specific models. These interlinked models form the basis for the consistency management and orchestration of the concretization process. In particular, we need to detect changes in the domain models which are conflicting with the domain-spanning system model and are likely to be relevant to other domains. These changes need to be propagated to the domain-spanning system model first and then to the other domain-specific models.
In the remainder of this paper, we use an example concretization scenario from the development of the RailCab system. Figure 4 shows the versions of the domain-spanning system model and domain-specific models that are created in the course of this example scenario as well as the propagation of domain-spanning relevant changes from one domain to other relevant domains.

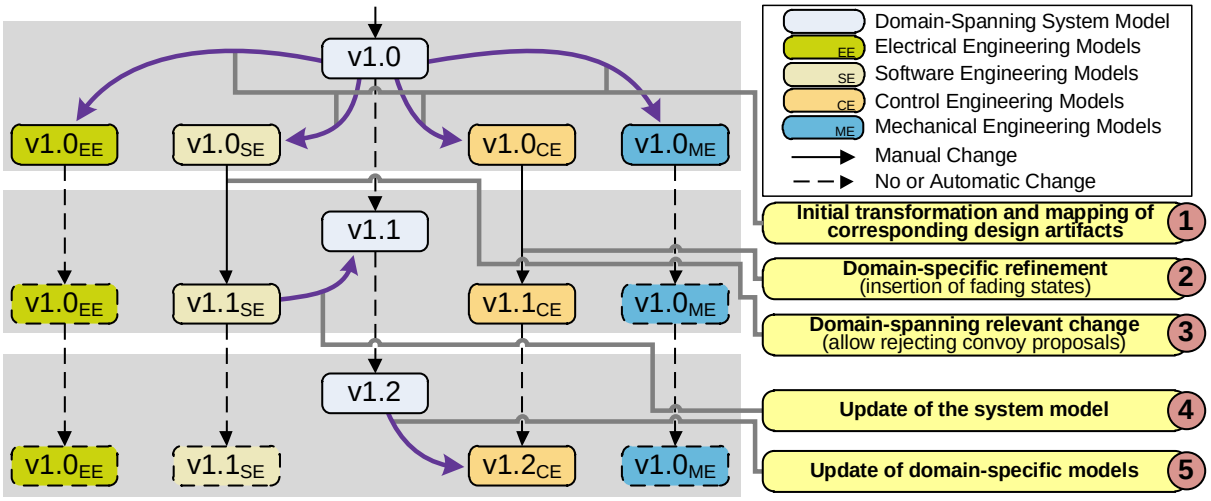

Figure 4. Propagation of relevant changes between the domain-specific models and the domain-spanning system model

First, we derive initial domain-specific models from the principle solution (marked with 1 in Figure 4). This can be done automatically using model transformation techniques. In our previous work, we presented such a transformation, focusing on the structural aspects of the system, e.g., transforming the active structure to a software component model [Gausemeier et al. 2009b].
In the course of the further concretization, the domains work concurrently using their domain-specific methods and tools. In particular, they extend and refine their domain-specific models. For instance, the control engineers (CE) insert additional fading states in which the actual transition between the control

strategies is performed (2). This is done for stability reasons, i.e., to avoid an abrupt switch between the controllers. This change does not affect other domains, as long as the fading time does not exceed the timing constraint specified in the principle solution (see Figure 3 c)). Thus, no update of the domain-spanning system model shall be performed.
The domain software engineering (SE) refines their models as well and performs a number of analysis tasks. In this example, it turns out that the initially specified behavior for forming a convoy does not consider the case that the leader RailCab may need to reject a convoy proposal for safety reasons, e.g., if the leader RailCab transports dangerous cargo. Thus, the software engineers extend the convoy communication protocol by modifying the corresponding statechart (3). As this modification may affect other domains, it is propagated to the principle solution (4), using model synchronization methods that we described in our previous work [Gausemeier et al. 2009b].
These changes in the behavioral model are also important for other domains; e.g., in control engineering, the switching between the control strategies must be adapted. However, the changed statechart cannot be simply copied to the control engineering model, as this model has already undergone some changes (the addition of the fading states) and these changes would be overwritten otherwise. The difficulty here is to update the control engineering model so that it reflects the changes from the software engineering, but in a way that the domain-specific refinement that happened before are not overwritten (5).
To sum up, there are two main challenges in this scenario. First, we have to specify which changes are domain-spanning relevant and which are refinements. To address this, we present a technique to formally define how domain-specific refinements look like: Domain experts create a set of so called *refinement rules* where each rule captures a general refinement operation. This rule set is then used by the model synchronization engine to detect whether a change is a domain-specific refinement, that must not be propagated, or a change is domain-spanning relevant.
Second, when domain-spanning changes are propagated to another domain-spanning model, domain-specific refinements in that target model may be overwritten, as they are not contained in other models. To address this issue, we describe how our model synchronization engine avoids the loss of such domain-specific information by reusing elements.

3. Behavioral consistency management

In the following, we describe in detail how we address these two challenges. Figure 5 shows the different behavioral models that evolve during the example scenario (the Stateflow and MechatronicUML models have been simplified for presentation purposes).
First, domain-specific models are generated from the principle solution using automatic model transformation techniques (marked with 1 in Figure 5). For software engineering, a MechatronicUML model is generated which contains a Real-Time Statechart that specifies the behavior for the convoy management (1 left). For control engineering, we generate initial MATLAB/Simulink and Stateflow models, e.g., a Stateflow chart for the convoy management (1 right).
Next, the control engineers implement the controllers using MATLAB/Simulink. Furthermore, they modify the Stateflow model by incorporating additional states which describe the fading behavior during the switching of the controller configurations (2). Such a change is considered as a domain-specific refinement that does not affect other domains. Therefore, it must neither be propagated to the domain-spanning system model nor to the other domains. So the model synchronization engine has to know that it must not propagate the change.
We propose that domain experts define a set of so-called *refinement rules* that describe which kinds of changes to a domain-specific model are regarded as domain-specific refinements. A refinement rule formally describes a refinement by a precondition (left-hand side) and how this precondition is replaced (right-hand side). Figure 6 shows a refinement rule which defines that adding an intermediate state is a domain-specific refinement. It describes that a transition may be replaced by a combination of a transition, a state and another transition. In addition, it is specified by a constraint that the new state and transitions must not violate the maximum duration of the original transition.
This refinement rule covers the addition of the fading states. Using this rule, the model synchronization can now detect that adding the fading states is just a refinement. Thus, it would not

propagate the change. However, as described later, it is important to store the information that a refinement took place, i.e., that the transition "createConvoy/" in the system model (v1.0 in Figure 5) now corresponds to the transition-state-transition combination in the control engineering model (2).

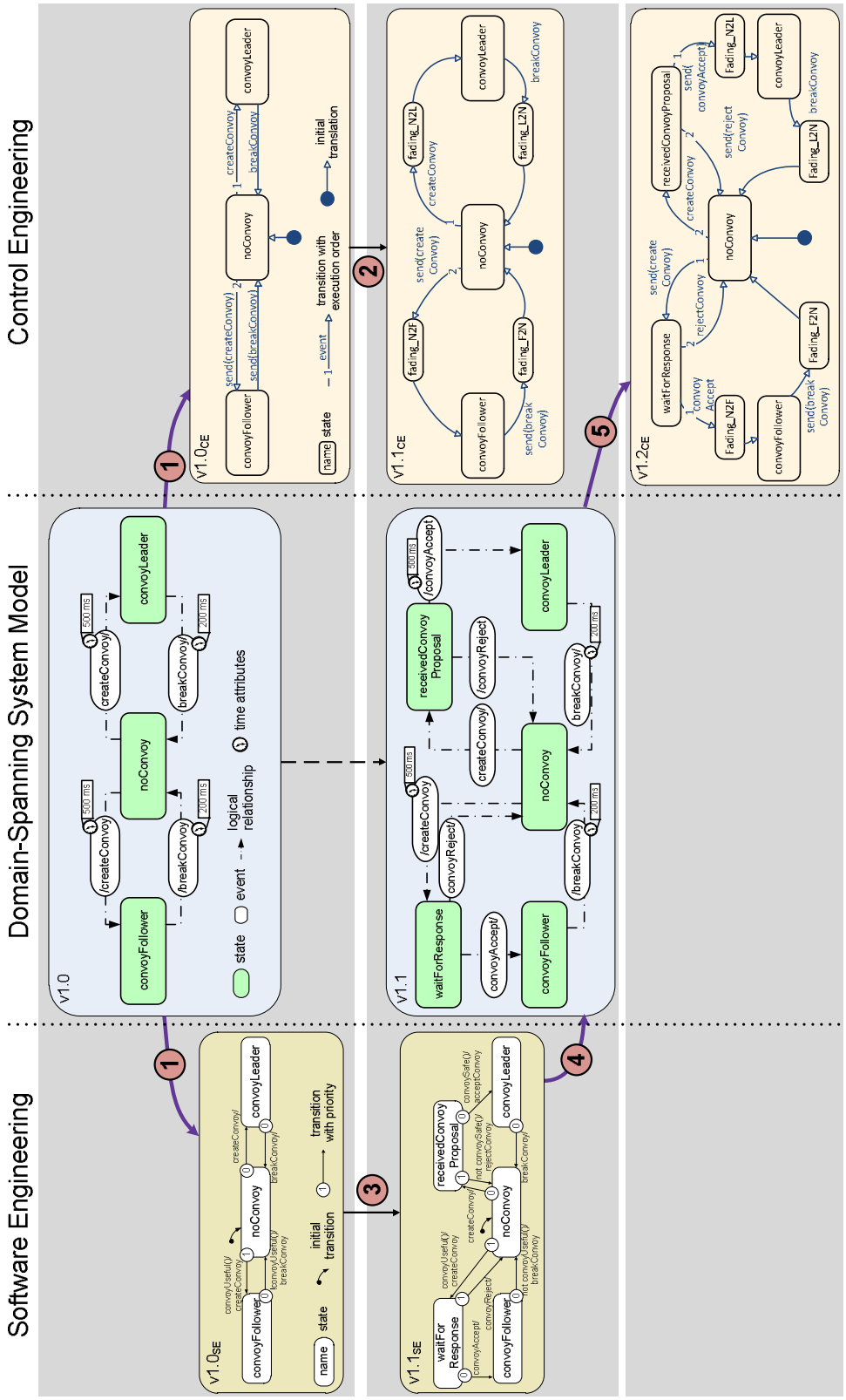

Figure 5. Evolution of the different behavioral models during the development process

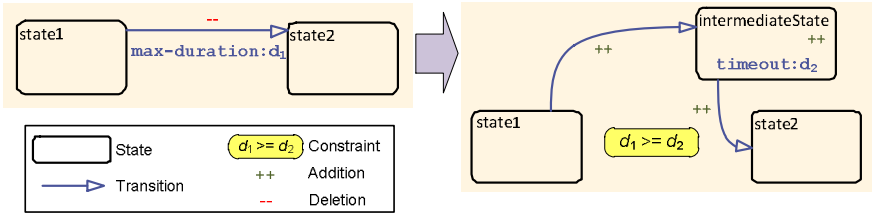

Figure 6. Refinement rule for adding intermediate states in the control engineering models

When choosing the language to define refinements, we sought to cover as many refinements as possible on the one hand and, on the other hand, not making the language too complex to make analyses impractical. We identified several different refinements from different domains (e.g., fault-tolerance patterns like triple modular redundancy, functional partitioning of components, load balancing) which can be described in terms of such rules. However, it remains to be investigated further whether we may need a more sophisticated language for other kinds of refinements.

At the same time, software engineers work on their model, too. They refine the behavior of the software by adding the possibility to reject a convoy proposal (3). The Real-Time Statechart is extended by two states "waitForResponse" and "receivedConvoyProposal" and new transitions and messages (see v1.1 of the software model in Figure 5). Instead of switching to the state "convoyFollower" directly after a "createConvoy" message is send, the follower RailCab switches to the new state "waitForResponse". There, it waits for the leader RailCab to accept or to reject the convoy formation proposal. The leader RailCab receives the "createConvoy" message and changes to the new state "receivedConvoyProposal", in which it decides whether it accepts or rejects the convoy proposal. If the convoy proposal is accepted, the leader RailCab changes its state to "convoyLeader" and the follower RailCab changes to the state "convoyFollower". If the convoy proposal is rejected, both RailCabs return to the state "noConvoy". This change in the discrete behavior affects other domains and must be propagated to the domain-spanning model and to other domains, as described in the following.

In general, model synchronization algorithms work in two steps: First, for everything that has been deleted in one model, the corresponding elements in the other model are also deleted. Second, for everything that has been added, new corresponding elements are created. When updating the domain-spanning system model to create version 1.1 (4), the automatic model synchronization does basically the same that happened in the software model: Amongst other things, it deletes the original transition "createConvoy" (from the "noConvoy" to the "convoyLeader" state) in the domain-spanning system model and creates a new state "receivedConvoyProposal" as well as three new transitions instead. The result is shown in Figure 5 (version 1.1 of the domain-spanning model).

The changes then must be propagated to other affected domains. Thus, the control engineering model has also to be updated to reflect the changed communication behavior (5). The challenge here is that the control engineering model has undergone some changes in the meantime (the addition of fading states). We have to assure that when propagating changes, no domain-specific refinement, e.g., the added fading states, is overwritten or disregarded.

Again, in the first step of the synchronization, for everything that has been deleted in the system model, the corresponding elements in the control engineering model are also deleted. In the example, the "createConvoy/" transition was deleted from the system model during the synchronization in (4). However, this "createConvoy/" transition from the system model corresponds to a refinement introduced in (2), i.e., the combination of the transition "createConvoy", the state "fading_N2L" and the transition to the "convoyLeader" state in the control engineering model. Thus, a naïve model synchronization approach would delete the complete refinement (see Figure 7 a)).

As such an information loss must be prevented, we propose an improved model synchronization approach. The main idea is not to delete such corresponding parts right away, but to *mark them for deletion* first, so they can be *reused later* [Greenyer et al. 2011]. Only unreusable elements marked for deletion are actually deleted. After propagating deletions by marking for deletion, we have to transform the added elements in the principle solution to the control engineering models. Previous model synchronization approaches would simply create new corresponding parts in the control

engineering model. Our improved synchronization tries to reuse elements marked for deletion instead: it performs a search in the set of elements marked for deletion and tries to reuse fitting elements; if they fit, they are not deleted. Only if no fitting previously deleted elements are found, new elements are created. For details of the improved synchronization algorithm, please refer to [Greenyer et al. 2011]. Refinements introduce information to a domain-specific model that is not covered by the domain-spanning model and the synchronization. Due to this additional information, there may be several possibilities to reuse previously deleted elements, which leads to differently updated models. All of these possibilities are "correct" in terms of consistency with the system model, but some may be more reasonable than others. In our example, the question is where the newly added states waitForResponse and receivedConvoyRequest should be added: before (Figure 7 b)) or after the fading states (Figure 7 c))? Of course, an expert can quickly see that c) is the correct way of updating, as the controller strategy must not be switch before every RailCab has actually approved the convoy. An automatic synchronization, however, cannot decide this.

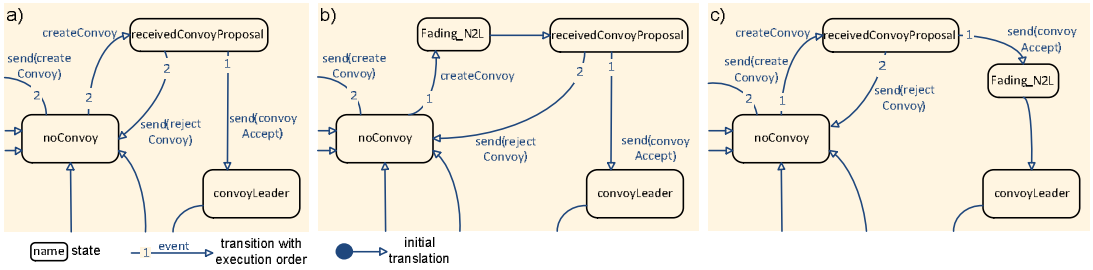

Figure 7. Cut-outs from Stateflow models updated in different ways: a) Lost fading state b) "wrong" propagation of the change, c) correctly updated Stateflow model

Thus, our improved synchronization algorithm explicitly computes all reuse possibilities, rates them with respect to information loss, and asks the user in ambiguous cases which of the update possibilities is the correct one [Greenyer et al. 2011]. In the example, the refinement in the control engineering model that has been marked for deletion (consisting of the transition "createConvoy", the state "fading_N2L" and the transition to the "convoyLeader" state) may be reusable as the corresponding control engineering part for three new transitions in the system model v1.1 ("createConvoy/", "/rejectConvoy", and "/convoyAccept"). However, the deleted refinement is not reusable as is. Some additional modifications have to be made to make it reusable in a certain case. For instance, when reusing elements marked for deletion as corresponding part for the new transition "createConvoy/" (which would result in Figure 7 b)), the target of the outgoing transition must be modified to point to the state "receivedConvoyProposal".

We can sort the different update possibilities by the amount of modifications that must be made to reuse the elements: the less modifications must be made, the more likely it is that this is a reasonable reuse possibility. In the example, we can reuse the refinement for the transition "createConvoy/" (see Figure 7 b)), as the source of the transition (the "noConvoy" state) is the same as before, but we must alter the target state. We can also reuse the refinement for the transition "/convoyAccept" (see Figure 7 c)), as the target of the transition is the same (the "convoyLeader" state). It is, however, unreasonable to reuse the refinement for the transition "/convoyReject", as neither the source nor the target state is the same as before. Thus, the user is asked which of the two reasonable reuse possibilities that are depicted in Figure 7 b) and c) should be used.

4. Concluding remarks

In this paper, we have shown how behavioral models are specified in the different domains during the development process of a mechatronic system. We have employed model synchronization techniques to automatically propagate changes that occur during the domain-specific concretization phase and are domain-spanning relevant. A central challenge here is that the models used are modeled on different levels of abstraction. That means that a domain-specific model may contain more detailed information than the domain-spanning system model; e.g., additional states may be added to a statechart. We

argued that a model synchronization must consider these refinements when synchronizing the models. As the main contribution of this paper, we described how such refinement possibilities can be formally captured using refinement rules, and we showed how the model synchronization must be extended in order to preserve such domain-specific refinements during subsequent propagations of changes from other domains. Usually, several alternative ways for updating the refined model exist. Our model synchronization approach calculates all alternatives, rates them, and asks the expert in ambiguous cases. We have illustrated the technique using behavioral models, but this technique can also be employed for refinements and consistency management of structural models. For instance, adding a triple-modular redundancy for a safety-critical sensor can also be seen as a refinement and captured by the refinement rules. We have implemented the extended model synchronization in our model synchronization tool, the TGG Interpreter[2]. Currently, we are developing the automatic integration of refinement rules into the model synchronization as well as the user interaction in ambiguous cases.
For future work, we plan on investigating how other techniques for multi-domain system development, such as DSM/Multiple-Domain Matrices, can be combined with our approach.

Acknowledgment

This contribution was developed and published in the course of the Collaborative Research Center 614 "Self-Optimizing Concepts and Structures in Mechanical Engineering" funded by the German Research Foundation (DFG). Jan Rieke is supported by the International Graduate School Dynamic Intelligent Systems.

References

Becker, S., Dziwok, S., Gewering, T., Heinzemann, C., Pohlmann, U., Priesterjahn, C., Schäfer, W., Sudmann, O., Tichy, M., "MechatronicUML - Syntax and Semantics", Technical Report no. tr-ri-11-325, Software Engineering Group, Heinz Nixdorf Institute, Paderborn, August 2011.

Friedenthal, S., Steiner, R., Moore, A. C., "Practical Guide to SysML: The Systems Modeling Language", Elsevier Science, 2008.

Gausemeier, J., Dorociak, R., Pook, S., Nyssen, A. Terfloth, A. "Computer-Aided Cross-Domain Modeling of Mechatronic Systems". Proceedings of the International Design Conference – DESIGN 2010, Dubrovnik, Croatia, May 17-20, 2010.

Gausemeier, J., Frank, U., Donoth, J., Kahl, S., "Specification technique for the description of self-optimizing mechatronic systems", Research in Engineering Design, Vol. 20, No. 4, 2009, pp. 201-223.

Gausemeier, J., Schäfer, W., Greenyer, J., Kahl, S., Pook, S., Rieke, J., "Management of Cross-Domain Model Consistency During the Development of Advanced Mechatronic Systems", Proceedings of the 17th International Conference on Engineering Design (ICED'09), Bergendahl, M.N., Grimheden, M., Leifer, L. (Ed.), Design Society, Stanford, 2009, pp. 1-12.

Greenyer, J., Pook, S., Rieke, J., "Preventing Information Loss in Incremental Model Synchronization by Reusing Elements", Proceedings of the 7th European Conference on Modelling Foundations and Applications (ECMFA 2011), 2011, pp. 144-159.

Henke, C., Tichy, M., Schneider, T., Böcker, J., Schäfer, W., "Organization and Control of Autonomous Railway Convoys". 9th Int. Symposium on Advanced Vehicle Control (AVEC08), October 6 - 9, 2008, Kobe, Japan, 2008.

Object Management Group, "Unified Modeling Language (UML) 2.3 Superstructure Specification", Document formal/2010-05-05, May 2010.

Pahl, G., Beitz, W., Feldhusen, J., Grote, K.-H., "Engineering Design – A Systematic Approach", 3rd English edition, Springer Verlag, London, 2007.

Stahl, T., Voelter, M., "Model-Driven Software Development: Technology, Engineering, Management", John Wiley & Sons Ltd., 2006.

Dipl.-Inform. Jan Rieke
Heinz Nixdorf Institute, University of Paderborn, Software Engineering Group
Zukunftsmeile 1, 33102 Paderborn, Germany
Telephone: +49-5251-60-3310
Email: jrieke@uni-paderborn.de
URL: http://www.uni-paderborn.de/cs/ag-schaefer

[2] TGG Interpreter, project web site – http://www.cs.uni-paderborn.de/index.php?id=tgg-interpreter