INTERNATIONAL CONFERENCE ON ENGINEERING DESIGN, ICED'09
24 - 27 AUGUST 2009, STANFORD UNIVERSITY, STANFORD, CA, USA

MANAGEMENT OF CROSS-DOMAIN MODEL CONSISTENCY DURING THE DEVELOPMENT OF ADVANCED MECHATRONIC SYSTEMS

Jürgen Gausemeier[1], Wilhelm Schäfer[2], Joel Greenyer[2], Sascha Kahl[1], Sebastian Pook[1] and Jan Rieke[2]
(1) Product Engineering, Heinz Nixdorf Institute, University of Paderborn, Germany
(2) Software Engineering Group, University of Paderborn, Germany

ABSTRACT
The development of mechatronic systems demands the close collaboration of engineers from different domains. In the course of the development, this leads to the creation of a number of separate, but interdependent models which capture the domain-specific system aspects. Without harmonizing the domain-specific development processes, inconsistencies between the domain-specific models are likely to occur. If these inconsistencies remain undetected, the system integration will fail, which is leading to increased development time and costs.
As a first step to prevent these problems, we propose a cross-domain system specification in the early conceptual design phase. Furthermore, as the novel contribution of this paper, we show how model transformation techniques can be employed to, firstly, derive initial domain-specific models for the subsequent domain-specific development and, secondly, to propagate domain-spanning relevant changes that may occur between those models. We show how the domain-spanning relevance of changes may be detected automatically and we discuss where expert decisions are indispensable. We implemented the approach in our development environment.

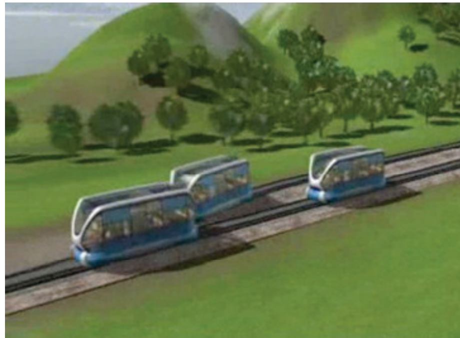
Keywords: consistency management, domain-spanning system specification, domain-specific models, model transformation, automated change propagation

1 INTRODUCTION
Advanced products of mechanical engineering and related industrial sectors often demand the close cooperation of mechanics, electronics, control engineering, and software engineering. This is aptly expressed by the term mechatronics. The rapid progress of information and communication technology opens up more and more fascinating perspectives for tomorrow's mechatronic systems: mechatronic systems with inherent partial intelligence. This technology is described by the term self-optimization. Self-optimizing systems are able to adapt their behavior to changing environmental conditions during their operation. During the development of the system the possible environmental conditions may not be fully known. For this reason, self-optimizing systems have to be capable to learn and optimize their behavior during the operation [1].
The development of mechatronic systems, especially self-optimizing systems, is still a challenge and is extensively researched at the Collaborative Research Center (CRC) 614. A particular challenge is to ensure the consistency among the domain-specific models of the system in the interdisciplinary cooperation between the different domains. To meet this challenge, a domain-spanning specification technique for the interdisciplinary *conceptual design* has been introduced. The result of this phase is a domain-spanning system specification that has to be refined by the different domains in the subsequent development phase. We call the domain-spanning system specification the *principle solution* and the subsequent, domain-specific development phase the *concretization*. During the concretization, however, it becomes difficult to assure the consistency between the domain-spanning system specification and the different domain-specific models. Because of the complexity of these domain-specific models, the consistency management must be supported by automated techniques.
In this paper, a solution for maintaining the consistency between all the domain-specific models and the domain-spanning system specification is presented. We have implemented a software tool to

automatically transform models from the principle solution into domain-specific models. Furthermore, our tool allows the engineer to automatically propagate changes among domain-specific models during the concretization. The approach is explained by an example from the development of an autonomous, self-optimizing transport system called *RailCab* [http://www-nbp.upb.de].
In the next section, we introduce the specification technique that we propose for the domain-spanning conceptual design. Furthermore, we outline the development process for self-optimizing systems and discuss the challenges which arise during and after the transition from the domain-spanning conceptual design phase to the domain-specific development phase. In section 3, the approach for automated consistency management is introduced. We show how the initial models for the different domains are derived semi-automatically based on the principle solution and we explain how the consistency management between these domain-specific models can be supported by automatic techniques for propagating changes among the domain-specific models. At last we conclude and give an overview of our future research.

2 DEVELOPMENT OF ADVANCED MECHATRONIC SYSTEMS

Neue Bahntechnik Paderborn/RailCab is a research project at the University of Paderborn for an innovative rail system. The core of the system consists of autonomous vehicles (RailCabs) for transporting passengers and goods according to individual demands rather than based on a timetable. To increase the capacity and to reduce the energy consumption, the RailCabs may autonomously form convoys. The RailCabs are driven by an electromagnetic linear drive that is located in its flat floor pan to which the different cabins for passengers and cargo are attached. Figure 1 shows an illustration of the RailCab and their capability to form convoys autonomously. A test facility on a scale of 1:2.5 has been built at the University of Paderborn.



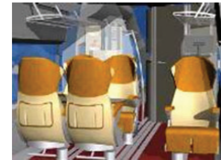


Figure 1. Illustration of the project “Neue Bahntechnik Paderborn/RailCab”

2.1 Challenges in the development of advanced mechatronic systems

The increasing complexity of mechatronic systems requires new development methods and tools. For one reason that is because these technologies are based on the close integration of mechanics, electronics, control engineering, and software engineering. In conventional development processes, the system is specified by a list of requirements. It is obvious that these requirements are only a rough specification of the overall system and thus leave much space for interpretation. For this reason there is a gap between the list of requirements and the integration of the domain specific models in the development phase *system integration* (see Figure 2). This gap can cause problems, since it is likely that the system integration fails due to different assumptions made in the development of the different domains.

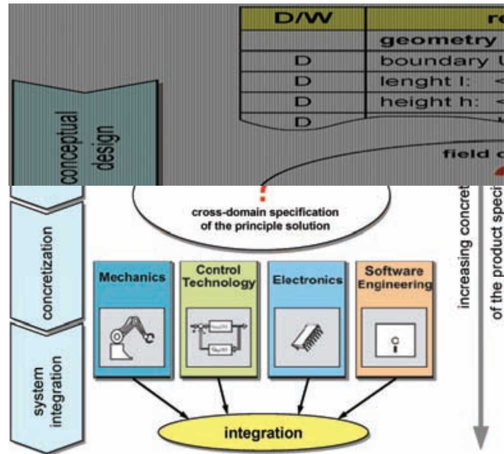


Figure 2. Central challenge: a new specification technique for the description of the principle solution of a mechatronic, respectively self-optimizing system

To close this gap and to prevent these problems, the involved domains have to cooperate throughout the conceptual design. The result of this early development phase is the jointly created principle solution. For the specification of mechatronic systems, a lot of different modeling and specification languages are available, e.g. Block Diagrams, Modelica, Function-Oriented Development (FOD) or UML/SysML. However, these specification and modeling languages do not fully meet the requirements to a domain-spanning specification language, particularly not for the design of self-optimizing mechatronic systems. For this reason, a new domain-spanning specification technique for the description of the principle solution has been developed in the context of the CRC 614. This specification technique is based on the research of FRANK, GAUSEMEIER and KALLMEYER [2]. For a holistic description of the principle solution of a complex system it is necessary to provide different languages for specifying different aspects of the system. According to Figure 3 these aspects are the requirements, the environment, the system of objectives, functions, the active structure, the shape,

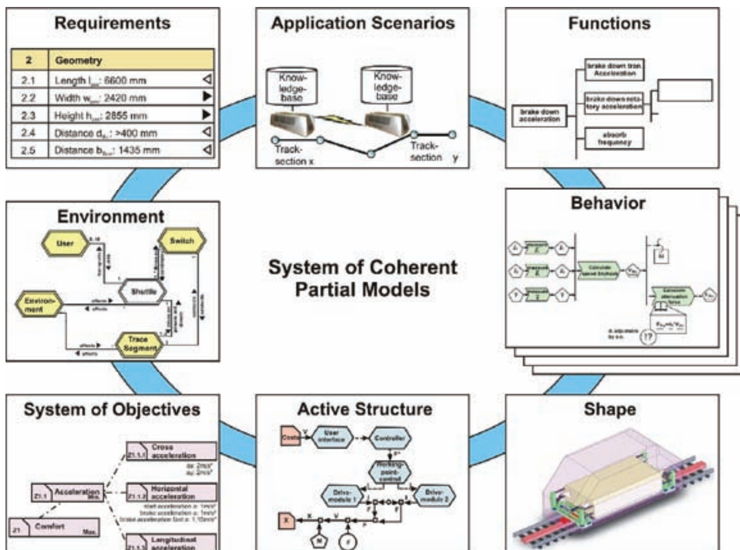


Figure 3. Partial models for the domain-spanning description of the principle solution of self-optimizing systems

application scenarios and the behavior. There are again various kinds of behavioral aspects, which need to be described, like the dynamic behavior of a multi-body system, the cooperative behavior of system components, electromagnetic compatibility, etc. There are also relationships between the aspects. The different aspects are represented by a system of coherent *partial models*. For details we refer to [3].

For these partial models and their relationships, a common meta-model is currently being developed that is the basis for a number of graphical editors. First prototypes of these editors have been implemented. Furthermore, this meta-model allows us to employ the automated transformation and consistency management techniques that are presented in this paper. In the following, we explain an example specification that is used to illustrate the consistency management concerns in the subsequent sections. Figure 4 shows part of the active structure of the principle solution of a RailCab with the system elements responsible for the velocity control and the formation of convoys. The system elements for the velocity control are grouped in the composed system element "velocity control". The control strategy for the velocity is reconfigured based on the current convoy state [4]. When the shuttle is in a convoy, the velocity is controlled based on the distance to the leading shuttle. The velocity control produces the reference value of the force F^* that the operating point controller controls on the traction unit. For details on the RailCab's control strategy, we refer to [5].

To form convoys and to control the velocity in a convoy, the RailCab has to interact with the adjacent RailCabs of the convoy. RailCabs can take the roles of the leader or a follower. The communication is realized via the system element "RailCab-to-RailCab communication module". Furthermore, the RailCab has to communicate with the system element "track section control" to synchronize the rotor and the stator of the electromagnetic linear drive. This is done via a second communication module that obtains the relevant position information from a position observer. The decision whether a convoy is formed or not is made by the system element "configuration control". Additionally, the configuration control is responsible for reacting to identified failures and hazards. The detection of errors and hazards takes place in the system element "hazard detection". One important aspect for the safety of the convoy is maintaining the safety distance d_{safe} between the RailCabs. To assure this distance, the hazard detection observes the position and velocity of the RailCab and its leader. The information about the position and the velocity of the leader is communicated over a wireless connection via the RailCab-to-RailCab communication module.

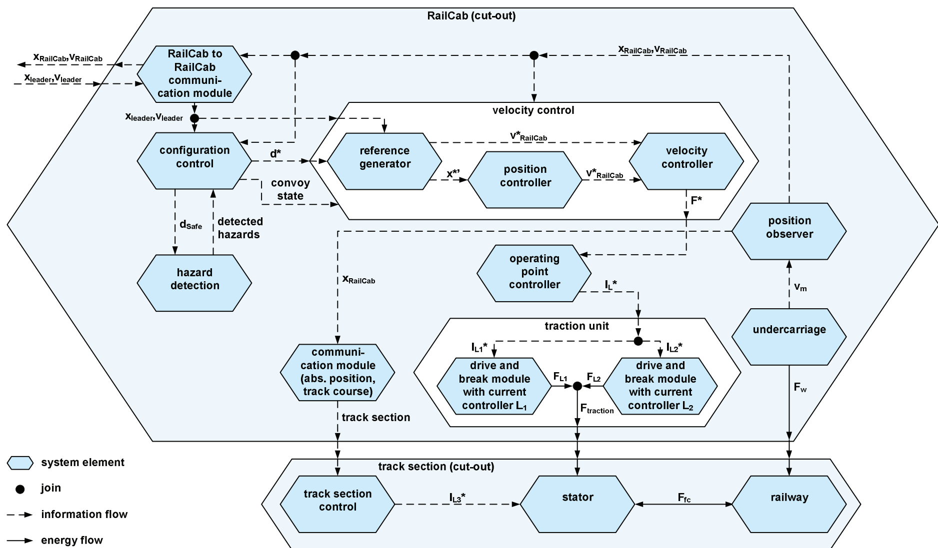


Figure 4. Part of the active structure of the principle solution of a RailCab

2.2 Consistency management in the domain-specific concretization

During the domain-specific concretization, which follows the conceptual design phase, it is crucial that the domain-specific models are refined and extended in such a way that they remain consistent with the domain-spanning system specification. A first step towards the consistent domain specific development is an automated transition from the principle solution to the domain specific design artifacts [6]. However, in the further course of the domain-specific development, changes may occur that are relevant for other domains. In this case, it is important that these domain-spanning relevant changes are detected and propagated between the domains. The consistency management is therefore an important aspect of the concretization process.

Figure 5 illustrates the development process: After the domain-spanning conceptual design, the domain-specific concretization takes place for all domains in parallel. However, the domain-specific activities sometimes need to be coordinated and synchronized. This especially applies whenever a domain-spanning relevant change takes place.

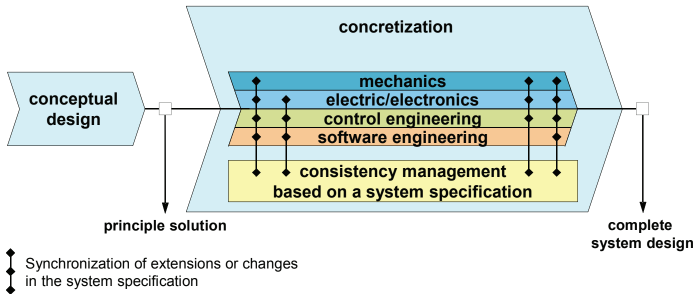


Figure 5. Synchronization of domain-specific models with each other and with the system specification

The common base model for all domains at the beginning of the concretization is the principle solution. In the further course of the development, we maintain an updated domain-spanning system specification as well as data structures which map the corresponding model structures in the system specification to the elements in domain-specific models. These interlinked models are the basis for the coordination of the concretization process. In particular, to assure consistency, we need to

- detect changes in one domain which are conflicting with the domain-spanning system specification and which are likely to be relevant for other domains
- propagate those changes to the domain-spanning system specification as well as to the other domain-specific models

Figure 6 schematically shows the versions of the domain-spanning system specification and the different domain-specific models that are created in the course of development. It also shows how changes in one domain that are relevant to other domains are propagated to these domains. In the following, we present a short change scenario based on the concretization of the RailCab example specification described above. The conceptual design phase is completed with the principle solution which serves as a basis for the domain-specific development. This version is numbered with “1.0” in Figure 6. The active structure shown in Figure 4 is part of this principle solution. A possible concretization scenario is the following.

1. The initial domain-specific models are generated from this principle solution and the domains start their domain-specific development [6]. The domain software engineering (SE) refines their models and performs a number of analysis tasks, for example a hazard analysis. It turns out that when the wireless connection fails between two RailCabs, the minimal distance cannot be guaranteed and collision accidents may occur.
2. One way to improve the safety of the system is to add an additional distance sensor. Let us assume that the software engineers change and validate their software model and now trigger a domain-spanning change process by proposing their solution to the other domains. Note that the software engineers may not specify the exact nature of the distance sensor; they only specify that there has to be some distance sensor component with a particular interface.
3. Next, the insertion of an additional distance sensor is propagated to the principle solution.

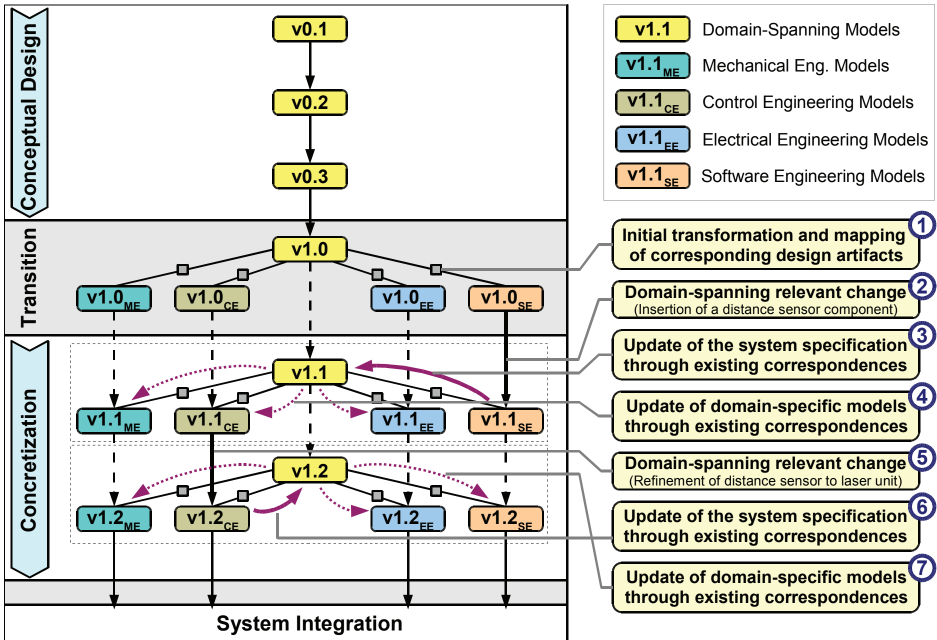


Figure 6. Propagation of relevant changes between the domain-specific models and the domain-spanning system specification

4. From there, the change is propagated to the other domains, since other domains need to adapt and refine their models based on this change.
5. Since the distance sensor is yet an abstract component, the control engineering decides to build in a laser-based distance scanner to measure the distance to objects in front of the RailCab. This refinement may include that additional properties of this particular sensor component are specified. Also, additional control blocks need to be added for signal preprocessing, etc.
6. A number of these refinements conducted by the control engineering are relevant for the other domains. Therefore, some changes are reflected back into the domain-spanning system specification.
7. Based on the updated system specification, the other domain-specific models are updated. Software engineering may now need to update the interface of the distance sensor, because particular properties of the interface became available. Also, the other domains have to adapt their models. Electrical engineering, for example, needs to extend the bus system and provide the energy supply for the laser unit. Mechanical engineering may need to adapt the shape of the RailCab in order to place the laser unit in the front of the RailCab's hull.

In the next section we show how automated model transformations can be employed to support change scenarios such as shown above.

3 MAINTAINING CONSISTENCY BETWEEN THE DOMAIN-SPANNING SYSTEM SPECIFICATION AND THE DOMAIN-SPECIFIC MODELS

Figure 7 shows part of the active structure diagram as presented in Figure 4 as well as the component diagram which is derived from it by an initial model transformation (1). The component diagram is an extended form of UML component diagrams, called *Mechatronic UML*, that was developed in the CRC 614 [4]. The figure also shows the additional sensor component added by software engineering (2) and the updated system elements in the active structure (3). (Note that the numbering in Figure 7 corresponds to the numbers of the change steps in Section 2, Figure 6.) In the following, we explain what information the user has to provide in this transformation/change scenario and we sketch how the transformation engine is working in the background to propagate relevant changes automatically.

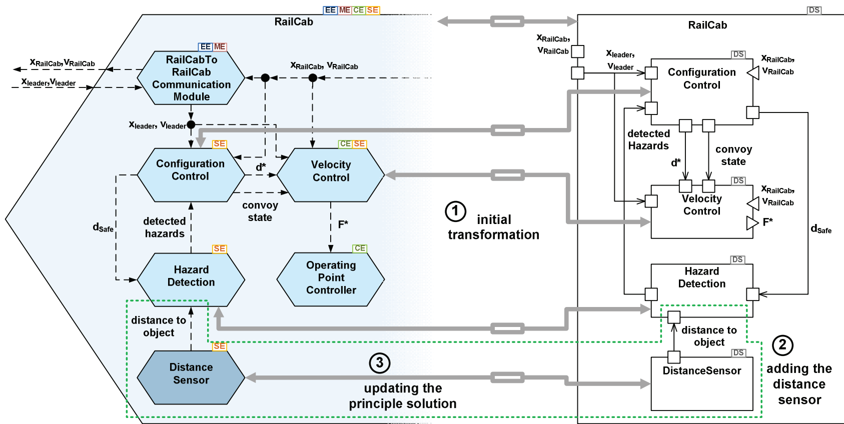


Figure 7. The initial transformation from the active structure into a component diagram (software engineering) and propagation of the additional sensor component

Before an initial transformation can take place, it is necessary that elements relevant for a particular domain are annotated in the principle solution. This is indicated in the active structure by the small labels on top of the system element hexagons. We see that each element annotated with "SE" (meaning "relevant for software engineering") is translated into a component in the component diagram and information flows are translated into connectors between the ports of the components. We may also annotate the information flows, properties or groups of elements in the active structure or the domain-spanning models. In addition to the annotations in the active structure, also the components in the UML diagram are annotated whether they are domain-spanning relevant or not. Since all components initially derived from the principle solution are relevant on the system specification level, the components in the component diagram are automatically annotated with the "DS" labels (meaning "domain-spanning relevant"). In the further course of the development, the domains may insert additional sub-components that are not marked relevant to other domains.

Between the active structure and the component model, there is an additional model, which preserves the mapping between corresponding elements. We call this the *correspondence model*. For the correspondences shown above, we have to ensure two constraints. We have to ensure that (1) each (SE)-annotated system element corresponds to a (DS)-relevant component and that (2) corresponding elements have corresponding parent elements. In the following, we call such correspondence constraints *relations*.

Relations as mentioned above can become very complex. We do not only have to define the relation between system elements and components, but also between ports, flows, connectors, properties, data types, etc. When we have to adapt our approach to other domain-specific modeling tools, we furthermore have to deal with the heterogeneity of the models. Heterogeneity is caused by the particular, tool-specific abstract representation of the models. Moreover, the relation between elements in the system specification and elements in the domain-specific models is not always a simple one-to-one relationship. For example, there may be cases where a system element is not relevant for a certain domain, but flows that "run through it" are. See the "RailCab-to-RailCab communication module" shown in Figure 7. This element is solely a hardware component that converts information that is conveyed by radio waves (position and speed of the leader RailCab in a convoy) into the same information in the form of electric currents on a bus. In the software model, however, this communication module is not relevant and we want to reduce it simply to a connector which delegates messages from the port of the RailCab component to the port of the configuration control component. To express this relation, the annotation data structure introduced above is in fact more complex than indicated by Figure 7.

To automatically analyze and enforce the relations between certain model patterns, it is not sufficient to specify the correspondence relations informally as sketched above. We need a formal specification of these relations that can be operationalized to aid in the consistency maintenance among the models. Such formalism should support a problem-oriented description of the relations that can be easily

maintained and extended by developers of an integrated development environment. We use a rule-based formalism called *Triple Graph Grammars* (TGGs) [7] by which we are able to capture exactly such relations between model structures as stated above. TGGs are very similar to a new model transformation standard by the Object Management Group (OMG) called Query/View/ Transformations (QVT). However, we do not use QVT because we have identified semantic problems with QVT. We have shown that TGGs can provide a formal basis for QVT and that QVT can be implemented by TGGs [8]. Furthermore, the existing tool support for QVT today does not support the change propagation scenario which we present in this paper.

Approaches exist where, for example, models for the design of chemical plants are translated to more specific models by TGGs and those models are automatically kept consistent to each other in the case that changes occur [9]. Also, TGGs have been applied for mapping customized SysML Parametric Diagrams to Modelica models [10]. We have implemented software tools that allow us to specify TGG rules graphically and to automatically interpret these rules for transformation and change propagation. In the following, we introduce TGGs in more detail.

A TGG consists of a set of rules that describe how corresponding graphs can be produced. A TGG rule links together graph production rules that describe how to create different kinds of graphs. Between these two graph production rules, a TGG inserts a third graph production rule to explicitly capture the correspondences of the graphs produced by the two other graph grammars. Since models can be considered graphs, we can use TGGs to describe how to create corresponding models by a stepwise application of the TGG rules. But, instead of creating arbitrary corresponding models, TGGs can be interpreted for different purposes, which we call *application scenarios*. One application scenario is called *forward transformation*: In case that only one (source) model is given, we try to determine a sequence of TGG rule applications that could have created this source model. Each time that we find a valid application of a TGG rule in the source model, we create the corresponding target and correspondence pattern accordingly, thereby producing the corresponding target model as well as the correspondence model. Another application scenario, called the *backward transformation*, works accordingly by reversing our notion of source and target. In case that two corresponding models were created by an initial transformation and changes occur to one of the models, we can interpret the TGG rules to propagate the changes made in one model to the other model. In this application scenario, called *update* or *propagation*, we make use of the existing correspondences between the models. The most general application scenario is called *synchronization* where TGG rules are interpreted to reestablish the consistency of two models in case that both models were changed concurrently. Our transformation engine currently supports the transformation and update application scenarios.

Let us have a closer look at the TGG rule in Figure 8. The rule consists of five columns that represent model patterns from different kinds of models. We see model patterns from the active structure, the UML component diagram, their annotation models and we see model patterns from the correspondence model. In the active structure column, there is a model pattern for a package which contains two system elements by the "packagedElement" relationship. One system element owns an instance of another system element via the relationship "ownedProperty". This is a situation similar to the active structure shown in Figure 7 where the system element RailCab owns a child element, for example a distance sensor. Similar to UML and SysML, we distinguish between the system element definition and its occurrence as a child element inside another system element. The pattern on the UML component diagram column expresses a component structure accordingly.

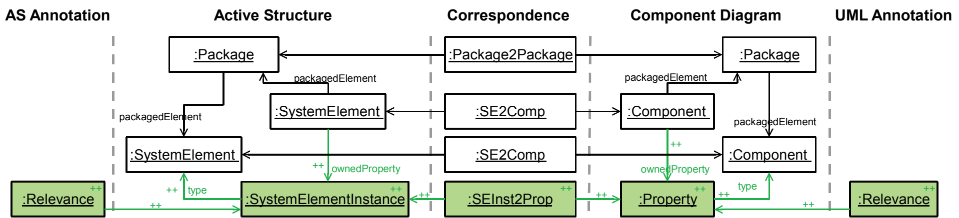


Figure 8. The TGG transformation rule for mapping domain-spanning relevant system elements to domain-spanning relevant components

The patterns in the columns consist of nodes and edges that are displayed in green and labeled with (++). We call these *produced nodes* and *produced edges*. Additionally, there are edges and nodes without the (++), which we call *context nodes* and *context edges*. The pattern constituted by the produced (resp. context) nodes and edges is called the *produced pattern* (resp. *context pattern*).

In the following, we explain how TGG rules as shown in Figure 8 can be interpreted by an algorithm to transform one model to the other (the forward transformation scenario). We recall that a TGG rule indicates that, whenever a structure in our corresponding models is found that matches the context pattern of a rule, the produced pattern in all models may be created "in parallel". This results in an extended set of models that are in a valid correspondence and further rules can be applied. In the forward application scenario, we need to match the TGG rule in the following way: First, we match the context pattern of the TGG rule in the existing source, correspondence, and target model. On doing so, it is important to check whether these objects have already been matched or created by prior applications of a TGG rule; we say that they were *bound* to the nodes of a previously applied rule. After matching the context pattern of the TGG rule, we have to match the produced pattern of the source side of the TGG rule in the source model. Now, in contrast, we have to make sure that these source model objects have not been matched by a prior rule application, i.e. they are *unbound*. On a successful match of the context pattern and the source produced pattern, we can create target and correspondence objects according to the target and correspondence produced pattern of the rule.

The rule in Figure 8 can be applied to transform a system element instance that is contained in another system element into the instance of a component that is the owned property of a parent component. The context structure was created by previous rule applications and edges between the produced nodes and the context nodes ensure that the new target model elements are created in the correct context. Additionally, the rule requires that there is a relevance object from the annotation pointing to the system element instance. On the UML side, a relevance object is created, which points to the corresponding property of the component. These relevance objects represent the relevance annotations that are displayed graphically in the diagrams. This way, we initially transform domain-specific models from the principle solution. Of course there are further rules for transforming attributes, ports and information flows. Also omitted in Figure 8 is that the rule contains additional constructs for specifying constraints on the attribute values of the model elements, for example to ensure that a component's name is equal to the name of the system element. Note that this form of TGG rules does not only specify the relation between two, but four models, because the annotation models are separate models. Therefore, we also call such TGG rules Multi Graph Grammar rules [11].

When changes occur on either model, for example when a system element is renamed, a component is moved, or a relevance annotation is removed, then the correspondence conditions may be violated. The TGG rules can be used by an algorithm to find such inconsistencies and, when they occur, the TGG rules can be interpreted to automatically repair the relation. For example, when a system element is renamed, the name of the corresponding component is updated accordingly. When a component is moved to another parent, the corresponding system element has to be moved to another parent system element accordingly.

In the change scenario described above (see Figure 7), the software engineer inserts a new component and annotates its relevance for other domains. In this case, the TGG rule in Figure 8 is applied for the backward transformation of the component. Figure 9 illustrates a part of the model displayed in Figure 7 as an object diagram. The backward application works similar to the forward transformation scenario described previously: First, we match the context pattern of the rule. On doing so, we have to make sure that the objects are already bound (they have been matched by a rule previously). This is expressed by the double ticks. We see that we require the component object named "DistanceSensor" to be bound already. That is because this TGG rule for transforming the representation of a component (property) requires the previous application of a rule for transforming its component definition; we assume that this rule has been applied previously. Next, we match the property and relevance object by the produced nodes on the component diagram and UML annotation side of the TGG rule. In this case, we require that the objects matched to the produced nodes are not bound. This is illustrated by the single tick on these objects. Upon a successful match of these objects, we create the active structure, active structure annotation, and correspondence objects (and links) according to the TGG rule (see also step (3) in Figure 7). The newly created objects are annotated by a double plus symbol. Figure 9 also shows objects that are yet unmatched by this rule application. These are the objects representing

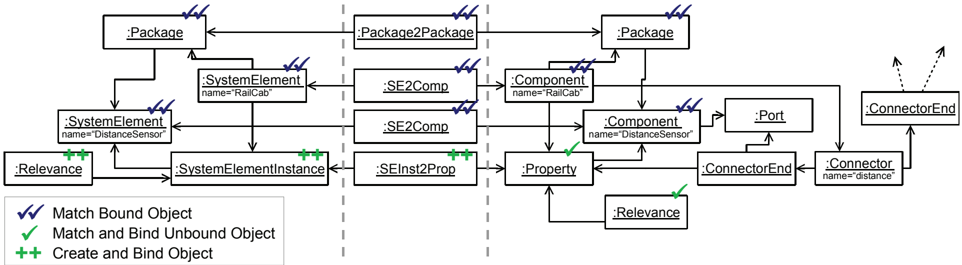


Figure 9. The object diagram representation of the change propagation shown in Figure 7

the port, connector, and connector ends. These objects have to be propagated by further applications of TGG rules until all objects have been processed by the TGG transformation engine.

A lot of inconsistencies among the models can be identified and propagated automatically by our transformation engine; however, there are changes which require additional input from the user. For example, a software engineer may decide to delete a component from the component diagram. By doing this, the mapping between the component and its corresponding system element instance is violated. The transformation engine may repair this inconsistency automatically in two ways: Either (1) the system element is removed from the domain-spanning model or (2) just the relevance annotation is removed from the domain-spanning model. This depends whether the user intends to (1) remove the component from the system model entirely or (2) whether he wishes to express that the component is not relevant for the software engineering model anymore. In this case the user has to interact with the transformation engine to determine how the inconsistency shall be resolved.

Furthermore, there may be changes in the domain-specific models where no algorithm can automatically determined whether those changes are domain-spanning relevant or not. For example, the software engineer may insert an additional software component for processing the distance sensor input. Normally, this is a domain-specific refinement and it does not have to be communicated with the other domains. However, this additional component may require more processing power or more memory has to be built in. This change would therefore require the coordination with the electrical engineering domain. In this case, the expert is indispensible for deciding whether a change is relevant to other domains or not. We may employ constraint solvers to automatically calculate whether the memory limit is exceeded by the software components deployed on it, but this again requires that an expert captures and formalizes the relationship between the memory of a processor and the memory required by software components. Still, even in cases where the expert decision is indispensible, our transformation engine helps to resolve inconsistencies by automatically propagating the relevant changes among the domains.

4 IMPLEMENTATION OF THE TRANSFORMATION ALGORITHM

We implemented the concepts explained above in a software tool, which currently supports the initial generation of UML component diagrams from an active structure and the propagation of changes between the component diagram and the active structure.

The tool is based on the *Eclipse* Platform [http://www.eclipse.org], which provides powerful open frameworks for building model-based development tools. Our tool consists of editors for the (domain-spanning) principle solution. Currently, there exist graphical editors for the partial models active structure, behavior-activities and behavior-states. See the left of Figure 10 for screenshot of the active structure editor. The meta-model for the partial models is based on UML 2.1. Furthermore, we have implemented the meta-models and the editor support for specifying the annotations for the active structure and UML component diagrams. Note that the annotation models are separate models and it is not required to modify existing modeling languages in order be able to annotate them. The plug-in mechanisms of the graphical editors in Eclipse allow us to visualize the annotations in the diagrams. See the active structure and UML component diagram in Figure 10. For editing UML component diagrams, we use the UML editors provided by the Eclipse UML2Tools project. This editor would need only minor extensions to support the editing of Mechatronic UML component diagrams.

For the model transformation, we use a transformation engine, which interprets Triple Graph Grammar rules for transforming models that conform to the Eclipse Modeling Framework (EMF).

Furthermore, the so-called *TGG interpreter* supports automatic bidirectional change detection and propagation. The TGG interpreter comes with a graphical editor for specifying the transformation rules (see the center of Figure 10). We also implemented additional plug-ins to provide user interfaces that support the user in change propagation tasks, for example when decisions must be made on how to handle inconsistencies.

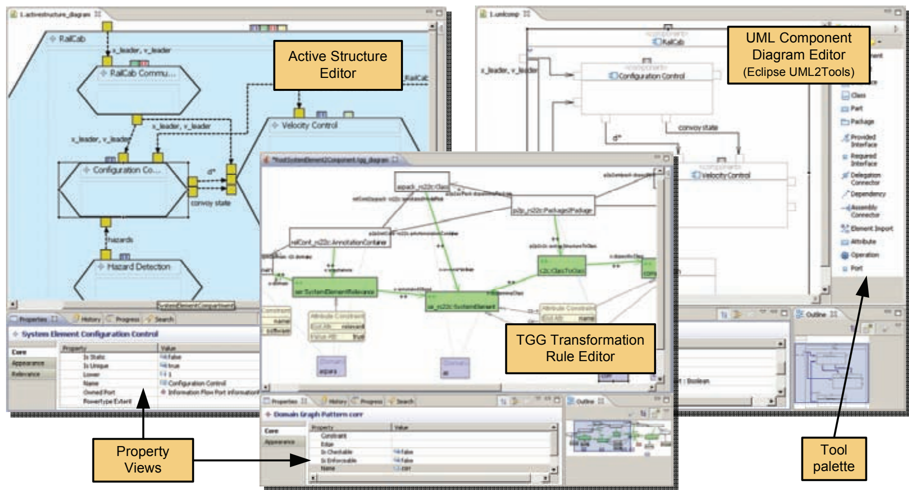


Figure 10. Screenshots of the Active Structure Editor, the TGG Rule Editor and the UML Component Diagram Editor

5 CONCLUSION AND FUTURE WORK

The particular challenge in the development of mechatronic systems is the collaboration of multiple domains. Typically, the different domains develop certain, domain-specific aspects of the system using their own, domain-specific development techniques and modeling languages. Therefore, it is likely that inconsistencies between the domain-specific models occur in the course of development. In this paper, we presented techniques for supporting a consistent development of mechatronic systems. We proposed the use of a domain-spanning modeling language for the collaboration of all domains in the early, conceptual design phase and presented an automated technique for ensuring the consistency of the domain-specific models, which are developed in the subsequent concretization. We have implemented our solution that consists of graphical modeling tools and an engine for transforming and updating models by the interpretation of a rule-based transformation specification. We have discussed to which degree the consistency of the overall system can be automatically preserved and where expert decisions are indispensible.

Currently, the presented transformation and change propagation techniques are applicable only to structural, hierarchical models like the active structure. In the future, we plan to investigate how these techniques can be extended to support the consistency management of behavioral models. Thus far, the TGGs only capture the syntactical consistency of models, but the rules are not powerful enough to consider the semantics of behavior models.

Furthermore, our approach yet needs to be extended to support a distributed development process. When large teams of engineers develop complex systems, conflicts between concurrent changes may occur that cannot be solved by our approach so far. We investigate how techniques for differencing and merging models [12], [13] can be combined with our transformation tool to support such development scenarios.

ACKNOWLEDGEMENT

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) under grant number SFB 614.

We also thank Christian Henke from the project Neue Bahntechnik Paderborn/RailCab for a very helpful discussion on the RailCab example.

REFERENCES

[1] Adelt, P.; Donoth, J.; Gausemeier, J.; Geisler, J.; Henkler, S.; Kahl, S.; Klöpper, B.; Krupp, A.; Münch, E.; Oberthür, S.; Paiz, C.; Podlogar, H.; Porrmann, M.; Radkowsky, R.; Romaus, C.; Schmidt, A.; Schulz, B.; Vöcking, H.; Witkowsky, U.; Witting, K.; Znamenshchykov, O.: *Selbstoptimierende Systeme des Maschinenbaus – Definitionen, Anwendungen, Konzepte*. HNI-Verlagsschriftenreihe, Band 234, Paderborn, 2008

[2] Gausemeier, J.; Frank, U.; Donoth, J.; Kahl, S.: *Spezifikationstechnik zur Beschreibung der Prinziplösung selbstoptimierender Systeme des Maschinenbaus*. Konstruktion, Teil 1 7/8-2008 und Teil 2 9-2008, Springer VDI-Verlag, Düsseldorf, 2008

[3] Gausemeier, J.; Zimmer, D.; Donoth, J.; Pook, S.; Schmidt, A.: *Proceeding for the Conceptional Design of Self-Optimizing Mechatronic Systems*. In: 10th International Design Conference, May 19 - 22, 2008, Dubrovnik, Croatia, 2008

[4] Burmester, S.; Giese, H.; Tichy, M.: *Model-Driven Development of Reconfigurable Mechatronic Systems with Mechatronic UML*. In: Model Driven Architecture: Foundations and Applications (Uwe Assmann, Arend Rensink, and Mehmet Aksit, eds.), Vol. 3599 of Lecture Notes in Computer Science (LNCS), pp. 47–61, Springer Verlag, August 2005

[5] Henke, Ch.; Tichy, M.; Schneider, T.; Böcker, J.; Schäfer, W.: *Organization and Control of Autonomous Railway Convoys*. 9th International Symposium on Advanced Vehicle Control (AVEC 08), October 6 - 9, 2008, Kobe, Japan, 2008

[6] Gausemeier, J.; Giese, H.; Schäfer, W.; Axenath, B.; Frank, U.; Henkler, S.; Pook, S.; Tichy, M.: *Towards the Design of Self-Optimizing Mechatronic Systems: Consistency between Domain-Spanning and Domain-Specific Models.* In: 16th International Conference on Engineering Design (ICED'07), August 28-31, 2007, Paris, France, 2007

[7] Schürr, A.: *Specification of graph translators with triple graph grammars*. In: Graph-Theoretic Concepts in Computer Science, 20th International Workshop, Vol. 903 of LNCS, Herrsching, Germany, pp. 151–163, 1994

[8] Greenyer, J.; Kindler, E.: *Reconciling TGGs with QVT*. In: Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems, MoDELS 2007, Nashville, USA, September 30 - October 5, 2007. (Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, eds.), Vol. 4735 of LNCS, pp. 16–30, Springer Verlag, September 2007

[9] Becker, S.; Herold, S.; Lohmann, S.; Westfechtel, B.: *A Graph-Based Algorithm for Consistency Maintenance in Incremental and Interactive Integration Tools*. In: Journal of Software and Systems Modeling (SoSyM), Vol. 6, No. 3, pp. 287–315, September 2007

[10] Johnson, T.; Paredis, C., Burkhart, R.: *Integrating Models and Simulations of Continuous Dynamics into SysML*. In: Proceedings of the 6th International Modelica Conference, Bielefeld, Germany, March 2008

[11] Königs, A.; Schürr, A.: *MDI - a Rule-Based Multi-Document and Tool Integration Approach*. In: Journal of Software and System Modeling (SoSyM), Vol. 5, No. 4, pp. 349–368, December 2006

[12] Mens, T.: *A State-of-the-Art Survey on Software Merging*. In: IEEE Transactions on Software Engineering, Vol. 28, Issue 5, pp. 449–462, May 2002

[13] Pottinger R.; Bernstein, P. A.: *Merging Models Based on Given Correspondences*. In: Proceedings of the 29th International Conference on Very Large Data Bases, Vol. 29, pp. 826–873, 2003

Contact: Prof. Dr.-Ing. Jürgen Gausemeier
Heinz Nixdorf Institute, University of Paderborn
Fürstenallee 11
33102 Paderborn
Germany
Tel: Int +49 5251 60-6267
Fax: Int +49 5251 60-6268
Email: Juergen.Gausemeier(at)hni.uni-paderborn.de