Dr. Wilhelm Schäfer is a professor and chair of the International Graduate School of Computer Science and Engineering at the University of Paderborn, Germany. Prior appointments have been with the University of Dortmund, Germany and McGill University at Montreal. He also spent four years in industry as head of the R&D department of a medium-size software house. His research interests are in object-oriented specifications for embedded systems, software processes, and re-engineering. He was program co-chair of the 23rd International Conference on Software Engineering 2001 and program chair of the European Software Engineering conference 1995. He is General Chair of ICSE 2008 in Leipzig, Germany.

Dr. Heike Wehrheim is a professor of Computer Science at the University of Paderborn. She obtained her PhD from the University of Hildesheim and her habilitation from the University of Oldenburg, both in Germany. Her research interests are in formal methods for the design of correct systems, in particular on specification and verification techniques. She is a member of the IFIP working group 6.1 and was program co-chair of its working conference FMOODS in 2006.
The Challenges of Building Advanced Mechatronic Systems

Wilhelm Schäfer and Heike Wehrheim
Institut für Informatik
Universität Paderborn
{wilhelm,wehrheim}@upb.de

Abstract

Mechatronics is an engineering discipline integrating the fields of mechanical engineering, electrical engineering and computer science. While the word “mechatronics” already has a long history, it is only the last ten years that we see their application all around us. Cars, CD players, washing machines, railways are all examples of mechatronic systems. The main characteristic (and driving force) of recent advances is the progressively tighter coupling of mechanic and electronic components with software. This makes software engineering (together with network technology) the main computer science discipline involved in mechatronics.

In this paper we survey current developments and discuss future trends in mechatronics, in particular from a software engineering point of view. The future of mechatronics will specifically see a move towards a high degree of adaptability and self-organisation. This poses new challenges on software engineering, especially on modelling, code generation and analysis. We exemplify existing as well as future strands by a collaborative research and development project of a mechatronic rail system from the University of Paderborn.

1. Introduction

Mechatronics is the engineering discipline concerned with the construction of systems incorporating mechanical, electronic and information technology components. The word mechatronics as a blend of mechanics and electronics has already been invented 40 years ago by a japanese company. Then, mechatronics just ment complementing mechanical parts with some electronical units, a typical representant being a photo camera. Today, mechatronics is an area combining a large number of advanced techniques from engineering, in particular sensor and actuator technology, with computer science methods. Figure 1 depicts the three areas of mechatronics and their overlap.

Typical examples of mechatronic systems are automotive applications, e.g. advanced braking systems, fly/steer-by-wire or active suspension techniques, but also DVD-players or washing machines. Mechatronic systems are characterised by a combination of basic mechanical devices with a processing unit monitoring and controlling it via a number of actuators and sensors (see also Figure 5, Section 3). This lead to massive improvements in product performance and flexibility. The introduction of mechatronics as a tight integration of mechanical, electronical and information-driven units allowed for turning conventionally designed mechanical components into smart devices. The significance of mechatronics is today also reflected in university education: mechatronics has become a degree on its own, and is at many places not merely taught by one area but jointly by all three. The subject managed to cross the traditional boundaries between engineering and computer science.

Today we see the first steps in the emergence of the next generation of mechatronic systems. While “intelligence” in the behaviour has so far always been achieved by gathering information (and reacting to it) from the one single machine, the usage and retrieval of information in the future will be characterised by an exchange of information between different machines. This can for instance already be seen in the automotive and rail domain: Intelligent lighting systems combine information about their environment obtained from their own sensors with those collected by
other cars. In the Paderborn rail system (introduced in more
detail in the next section) shuttles autonomously form con-
voys to reduce air resistance and optimise energy con-
sumption. This is a general trend: The smart devices of
today’s mechatronic systems will turn into “populations” of
smart devices, exchanging information for optimising their
local behaviour as well as possibly competing for limited
resources. This movement imposes in particular new chal-
 lenges on the computer science side in mechatronics. The
mechatronic systems of the future will be characterised by
the following properties:

• **High degree of concurrency**: Systems will consist of
a large number of autonomous components, exchanging
information while running in parallel. Components
may form clusters to collaborate on a common goal but
may also compete as to optimise their own aims.

• **Decentralisation**: Due to the high degree of concurrency
and distribution systems cannot be centrally ob-
served and as a consequence not centrally controlled.

• **Self-Coordination**: As a result of the previous two
points, advanced mechatronic systems will largely
have to rely on principles of self-coordination.

Several disciplines in computer science are affected by
this change. For achieving reliable and secure transmission
of information the areas of network technology and cryp-
tography are challenged. With respect to the issue of self-
coordination it is in particular software engineering which
has to make a major step towards a new design method-
ology. Current self-* developments in software engineer-
ing are already making steps in this direction. For
the design of complex mechatronic systems of the future
these have to be combined and complemented with other
advanced techniques as to form an engineering method
for self-coordinating systems. Such a method in particular has
to involve

• new modelling formalisms integrating model transfor-
mations (describing adaptation, reconfiguration etc.)
themselves into the model,

• new code generation techniques, operating at run-time
and taking platform specific parameters into account,

• elaborate formal analysis techniques being able to
cope with the high volatility of systems (and prop-
erties emerging by a continuous dynamic change).

In addition to the challenges that future mechatronic sys-
tems will bring to software engineering, there are also a lot
of unsolved issues in the design of current systems. While
mechatronic systems indeed incorporate parts constructed
by different engineering disciplines and computer science,
the actual cooperation during the construction is less de-
developed. There is no joint development process, no joint
tool usage, no joint modelling formalism and no joint anal-
ysis. Every discipline has its own approaches; an integrated
framework for the construction of mechatronic systems is missing.

In the following we will sketch a general architectural
model of mechatronic systems, exemplified by a mecha-
tronic rail system from the University of Paderborn. From a
software engineering point of view we survey current state
of the art in the development of mechatronic systems. This
especially covers modelling formalisms and tool support
as well as analysis techniques. The last part is devoted to
pointing out future developments and research challenges
which we believe characterise advanced mechatronic sys-
tems of the future.

## 2 A General Architectural Model for Mecha-
tronic Systems

As mentioned above, the Paderborn-based RailCab research
project (http://www-nbp.upb.de/en) is a concrete example
for a mechatronic system of the next generation. It aims
at combining a passive track system with intelligent shut-
tles that operate individually and make independent and de-
centralized operational decisions. The project is funded by
a number of German research organizations. It has built
a test track in the scale of 1:2.5 such that the ideas of the
project are not only tested "on paper" but in real operation
(cf. Fig. 2).

**Figure 2. The test track and shuttle prototype of the RailCab project**

The vision of the RailCab project is to provide the com-
fort of individual traffic concerning scheduling and on-
demand availability of transportation as well as individually
equipped cars on the one hand, and the cost and resource ef-
ectiveness of public transport on the other hand. The mod-
ular railway system combines sophisticated undercarriages
with the advantages of new actuation techniques as em-
ployed in the Transrapid (http://www.transrapid.de/en) to
increase passenger comfort while still enabling high speed transportation and (re)using the existing railway tracks.

One particular aspect is to reduce the energy consumption due to air resistance by coordinating the autonomously operating shuttles in such a way that they build convoys whenever possible. Such convoys are built on-demand and shuttles travel only a few centimeters apart from each other (up to 0.5m) such that a high reduction of energy consumption is achieved. This requires a lot of information exchange between the various machines or system components like the shuttles, registrars, dispatchers, stations, customers etc. It also means a tight integration of quasi-continuous and discrete control software and the realization of complex functionality by software rather than hardware, because, for example, travelling only at a few centimetres distance in a convoy requires tight coordination between the various speed control units under hard real time constraints.

A so-called Operator-Controller-Module (OCM) as depicted in Fig. 3 (cf. [21]) describes a general architectural model of a single machine or system component and identifies its constituent parts.

The OCM reflects the strict hierarchical construction of a system component including the hardware components: (1) On the lowest level of the OCM, a more or less standard controller (C) implements a feedback loop based on sensor input and producing actuator control signals accordingly. This is called the motor loop. The software processing is necessarily quasi-continuous, including smooth switching between possible alternative control strategies which are described by some form of differential equations or difference equations. (2) The controller is controlled by the reflective operator (RO), in which monitoring and controlling routines are executed. The reflective operator operates in a predominantly event-oriented manner and thus includes a control automaton with a number of discrete control states and transitions between them. It does not access the actuators of the system directly, but may modify the controller and initiate the switch between different control strategies. Furthermore it serves as the connecting element to the cognitive level of the OCM. (3) The topmost level of the OCM is called the cognitive operator (CO). On this level, the system can gather information concerning itself and its environment and use it for the improvement of its own behavior, i.e. possibly complex, time-consuming computations for long-range planning. This level introduces intelligent behaviour and consequently such components could also be called agents.

Figure 3. OCM architecture and its elements

The OCM hierarchy can be nested, i.e. each nesting level may include an OCM which however does not include the controller part. Controllers, which implement the continuous part of the software, usually exist only on the leaf level of a nested OCM hierarchy. As an example, consider the above mentioned shuttles of the RailCab project. The architecture is defined by OCMs w.r.t. the reflective operators and the controllers as depicted in Fig. 4. A shuttle consists of components like the suspension/tilt module, the engine, or the steering module. They will utilize the same hardware (actuators, sensors and controller) but their software is defined by OCMs as is the software of the shuttle.

As a complete mechatronic system usually consists of several concurrently running components, there exists a further possibility of communication between components besides the strict hierarchical control flow. Top-level OCMs of several nested hierarchies, which usually represent a major system component, may act as freely interacting agents. This means that agents exchange information but no cen-
Central control is defined anymore. As examples of such major system components consider the different shuttles, stations, job brokers and dispatchers of the RailCab project.

Such a system as RailCab exhibits all the features of a next generation mechatronic system, namely high degree of concurrency, no central control, the necessity for self-coordination, a possible reconfiguration of the system at runtime due to the volatility of system components, and the emergence of new system properties because of the cooperation of (intelligent) agents. It also faces challenges which have been only partly addressed appropriately by the development methods of state-of-the-art mechatronic systems like software running under hard real time constraints, the integration of continuous and discrete control units and a high demand for quality as such a system is often used in and built for safety-critical applications.

3 State of the Art

Modelling & Tools. In a certain sense, modelling and even model driven development, i.e. the generation of executable code from a model, has long been existing in the mechatronic world to improve software quality based on model analysis. Classical feedback controllers as sketched above, are specified by a combination of block diagrams and differential equations. Typically platform specific code is generated from such a model specification.

A feedback controller has to be designed in such a way that excitations and disturbances do not lead to oscillations which, in the case of a shuttle for instance might even result in a collision or derailment. If a controller holds the output on a desired value or within a given range even after an excitation or disturbance, the whole system is called stable.

In some cases, it might be necessary to switch between different controllers, i.e. exchange one controller by another one. This could lead to a (discrete) jump in the output signal which again might result in unintended oscillations. To avoid those also called unstable situations, a so-called output cross fading function is defined. This function specifies the time needed to fade out the output of the controller which is to be replaced while fading in the output of the new controller.

Many commercial tools support the specification of controller models by block diagrams and differential equations. In addition, they provide for simulation of the specified model, often in terms of viewgraphs, to support analysis of the model. Numerous solutions exist to generate platform specific code, often for very specialized hardware. However, this code generation is usually not flexible, i.e. the particular code generator is domain or mostly even target platform specific.

The de-facto standard tool, which is most widely used in many industrial applications, is Matlab/Simulink which is developed and commercialized by Mathworks.

However, modelling (and corresponding analysis) gets a lot more difficult when not only stand-alone controllers have to be developed but a usually complex network of those systems has to be built, as exemplified by the rail-cab system. Here, not only a shuttle in itself contains such a complex network but also the necessary connection and communication of shuttles, stations, brokers, users etc. forms such a network. In addition, each network component is more complicated than a single controller as indicated in the OCM hierarchy by the reflective operator and cognitive operator levels respectively. In fact, components now also exhibit discrete behaviour, maintain corresponding data structures to be able to learn from history and communicate extensively with their environment based on possibly complex communication protocols and not only by (simple) sensor input.

This situation made it necessary to introduce concepts for modelling discrete systems. (Static) component models and corresponding discrete behaviour have to specified.

Component models are usually represented by class or component diagrams resp., where the UML 2.0 component model and especially its extension the Systems Modelling Language SysML [31] could be considered now a de-facto standard to be used for this purpose. SysML is a response to OMGs request for a proposal of a UML for systems engineering. In SysML blocks specify the fine-grained structure of a class extended by continuous communication links between ports. Continuous components are specified by parametric constraints on class attribute values expressing corresponding differential equations.

Component models are also defined domain specific with AUTOSAR being one of the most prominent examples. AUTomotive Open System ARchitecture is an international project of leading car manufacturers and suppliers with the goal to develop and establish a standard architecture for electrical and electronic components in cars. It defines major architectural components and interfaces to enable the ex-

![Figure 5. The interplay between mechatronic components](image-url)
change of software (and underlying hardware) easily. The leading principle is "Cooperate on standards - compete on implementation".

More advanced concepts how to build the architecture or component model of even self-managed systems are described in [26].

Widely used approaches to model behaviour are Petri Nets and many kinds of finite state machines called state charts, state flow diagrams and sometimes even data flow diagrams. Except the most notable definitions of timed automata and timed Petri Nets, they usually lack in describing especially timing constraints which are needed to model and analyse time critical applications as mechatronic systems.

Again many commercial tools, especially from the software engineering domain like Rational Rose, support modelling the component structure and discrete behaviour.

Tools from the engineering domain like Matlab and in particular its extension by Stateflow also support modelling discrete behaviour however in a very limited sense. Only support for modelling simple state charts is provided but no support for defining component structures and especially no support to define time constraints is available.

The key issue is however to model discrete and continuous behaviour in a uniform approach also including the modelling of the usually complex component hierarchy as exemplified by the following example taken from [13].

![Figure 6. A sample of a hybrid statechart](image)

In Fig. 6 the internal behavior of a Shuttle component is defined by a so-called Hybrid Realtime State Chart (HRTSC). As an example for a typical real-time requirement, a deadline interval $d_1$ is used to describe that the state change from state noConvoy to state convoyFront has to be finished within a given interval. Similarly, deadlines are defined to constrain the time an object may remain in a certain state. Guards of transitions may contain conditions which depend on the status of a clock. When defining the constraints which express the reconfiguration of controllers, like when switching between state convoyFront and noConvoy respectively, the above mentioned fading functions have to be considered and represent lower and upper limits for those constraints. A HRTSC is an extension of a Realtime State Chart where a certain controller configuration can be assigned to a particular state. In the example two different types of speed controllers exist which are used depending on the shuttle running in convoy mode or in non convoy mode including running at the top of a convoy. The Velocity controller controls speed based on some input by a user or dispatching system and the current speed of course, whereas the Distance controller has also to observe the speed of shuttles running in front of it.

**Code Generation.** Based on such a specification, model based development ideally requires the generation of code which meets all realtime constraints as defined in the model specification. This requires the code generator to know about all platform specific constraints like speed and number of processors or available memory.

Only a very few research oriented approaches exist to support a uniform modelling of the behaviour of all system components including the specification of realtime constraints and a corresponding code generation. [21, 25, 10, 2, 9] present first steps to provide for support of e.g. hierarchical modelling, reconfiguration of controller configurations as well as a smooth integration with the definition of static component structures and especially the specification conformant code generation.

**Processes.** The above description focussed on modelling the software part of mechatronic systems. One of the most prominent problems in current industrial development and even research approaches is however the lack of integration between the different disciplines, namely mechanical and electrical engineering and computer science or software engineering more specifically. The current situation is usually characterized as the "throw it over the wall" approach. Usually, the mechanical engineer starts with designing the shape and mechanical parts, then the electrical engineer plans the wiring and finally the software engineer has to write the code. Obviously, this approach leads to a lot of design errors and costly rework when it is finally noticed that some parts do not fit together or the simple layout of processors and memory make certain software solutions impossible.

**Analysis & Tools.** A rather large percentage of mechatronic systems are deployed in safety critical areas (e.g. the automotive or rail domain, see [29, 11, 5] for model-based developments in the automotive domain). This makes analysis of mechatronic systems (or first of all, their models) one of the main areas of work for software engineers employed in the design of such systems. Since its invention in the ’80ties model checking [15] has become a standard technique for verification, in particular for hardware systems. The main advantage of model checking which makes it interesting for mechatronic systems is its (almost) full automation, providing tool support for analysis. Notwithstanding recent advances and success stories, the main challenge is still the so-called state explosion problem: model
checking techniques (most often) rely on a search of the whole state space, and this can grow to arbitrarily large dimensions. Thus research in the area of automatic verification today focuses on fighting the state explosion problem (for a discussion of current approaches to software model checking see [18]). While in the past several standalone approaches have been developed (e.g. symbolic model checking with BDDs [16], the use of SAT solvers in bounded model checking [12], powerful reduction techniques like abstraction and partial-order reduction [3, 17, 27]), current work focuses on the tight integration of these techniques: SAT solvers are combined with decision procedures (giving so-called SMT-solvers, see e.g. [4]), model checking with specific AI search methods [19], bounded model checking is parallelised [1] or model checking combined with static analysis methods [6]. Prototype tools supporting these new developments are under development.

Mechatronic systems present a further challenge for verification as they belong to the area of hybrid systems, characterised by a combination of discrete and continuous parts. The software constitutes the discrete part, while the continuous dynamics corresponds to the physical system with its sensors and actuators. Verification of hybrid systems today is still in its infancy. Well understood is the subclass of timed systems, where the only continuous part is the change of time. System models in this class are written as timed automata, and a number of tools (most prominently Uppaal [8] and Kronos [34]) support verification of timed automata with respect to reachability or even temporal logic specified properties. For hybrid systems, analysis tools supporting particular classes of continuous dynamics (e.g. algebraic constraints or linear hybrid automata) are for instance HyTech [24] and CheckMate [30]. Automation can still only partially be achieved, the algorithms employed in the model checking are not guaranteed to terminate anymore. In order to make the actual system fit into the required subclass, approximations of the real system are used (with the risk of losing precision).

The approach to verification taken in the RailCab project embodies a combination of a number of different techniques. The RailCab model belongs to the class of hybrid systems. The basis for the verification is the structured UML-based model of the mechatronic system, dividing it into a set of components interacting through well-specified interfaces. Verification pursues a compositional approach: instead of analysing the system as a whole (which is not in the range of automatic methods), components together with parts of the environment are verified separately. The context of a component, i.e. the relevant part of the environment to be considered therein, can be determined by the specific role that the component takes on in the system. The results of these separate analyses can be combined into a verification result for the complete system using an assume-guarantee style [23]. For the verification of single components, which in particular embody controllers and thus continuous parts, a combination of model checking, abstraction and testing is used. Controllers are modelled as timed automata and thus only constitute abstractions of the continuous behaviour. The correctness of this abstraction is checked by the mechanical engineers who – besides having a rich knowledge of controllers - actually carry out a lab testing of the behaviour. Verification of a single component and its context is then automatically carried out using the timed automata model checker Uppaal.

There are some important prerequisites for this type of verification which we believe are fundamental for any type of analysis for mechatronic systems:

- a structured component-oriented model with well-defined small interfaces between components,
- a precise formal description of components (on a certain level of abstraction) and
- confidence in the abstractions taken (at the best supported by another analysis technique if unfeasible for model checking).

Still, this is not the end to verification. The analysis results are only transference to the actual realisation if the assumptions made in the model (for instance on port-to-port connections, communication times, etc.) are met in the physical realisation. This has to be guaranteed by on the one hand code generation and the other hand the mechanical and eletronical realisation.

4 Future developments and research challenges

We believe that future mechatronic systems will consist of several autonomously acting agents capable of monitoring their own physical environment as well exchanging information with other agents. The presence of this additional information will give rise to completely new possibilities of adaptation, which go well beyond what software control in mechatronic systems is currently achieving. This trend is further supported by the emerging new techniques in network technology, e.g. wireless adhoc networks. Constructing the software of such advanced systems requires a number of significant changes of current software engineering techniques. In particular, the following issues have to be addressed to build the next generation systems properly.

- Current software design processes are tailored towards a particular domain rather than spanning over all involved domains. So-called concurrent engineering is rather a goal than reality in practice.
• Modelling formalisms allow for a description of static systems but not for their volatility. Model transformations are meant for transforming models towards a particular use on a platform but not for describing the change that a model (viz. the modelled system) itself may undergo during its lifetime.

• Analysis techniques mainly rely on the knowledge about a global state space and cannot cope with properties only emerging due to the volatility of systems.

• Secure exchange of information is usually based on a central unit keeping public keys of participants and cannot manage decentralised highly distributed systems of agents dynamically building as well as resolving clusters. Communication in future mechatronic systems will however cross the borders of one agent and thus has to be secured.

While the latter is clearly a task for cryptography and network technology all of the former ones are challenges for software engineering, sometimes necessarily in cooperation with the other disciplines.

Processes. The existing "throw it over the wall" approach as the basis of a development process is of course a result of traditional development where the mechanical part of a system used to be the most complex and difficult one. Psychological barriers in the heads of developers, who do not want to loose the overall control, are one of the main reasons for the current situation but not the only one. Technically, a different and somewhat overlapping terminology is being used which often hampers common understanding. As an example just take the word version which in software engineering usually addresses the change of a piece of software over time. In mechanical engineering, version is a new product which has come out of a complex design and development process like the new version of a car. Consequently, related terms like variant and configuration are also used in slightly different meanings. Deeper technical issues than just harmonizing terminology represent further challenges. In engineering, so-called product data management systems (PDM) are the central repository to store and control the intermediate results of a development process whereas software version and configuration management (SCM) systems are used in the software area. The PDM systems are usually initially filled by a CAD system and thus their content is based on the definition of the constituent parts of a (new) product. If at all, software is attached to these product parts. PDM systems, despite offering some support for SCM, do not support the kind of cooperative work, frequent updates, automatic messaging, baselining etc. which SCM systems offer. On the other hand, SCM systems do not offer any PDM type of support, i.e. the storage of geometric shapes, the detailed product hierarchy etc. In fact, an integration of these very different lines of development for central repositories is one of the main short term goals to achieve to support true concurrent engineering.

Modelling. The foremost goal of modelling must be the uniform specification of the discrete and continuous parts of an advanced mechatronic system across all disciplines. This includes the usually deeply nested component hierarchy, the communication structure and behaviour of components as well as the corresponding behavioural specifications.

As those networks of nested components become very complex, such a uniform language must be defined in such a way that all possible syntactic checks to avoid errors must be exploited. This requires the definition of a precise static semantics (or context sensitive syntax) of the language. As much as possible, the grammar (or metamodel) of such a language must define precisely e.g. a type system of mechatronic components, a refinement notion or the conformance of required and offered component functionality. A type system, for example, defines the permitted connections of component ports whereas a refinement notion enables to check syntactically (at least to some extent) whether a component is correctly substitutable by a set of other components on the next lower hierarchy level. Of course, such a definition of refinement may not be too restrictive as to make it very complicated and cumbersome to model real world systems.

As the component definition must include hardware and software components, the modelling language becomes domain specific at least to a certain extent. Properties of sensors and actuators, their interfaces to software control (in fact to the reflective operator), or possibly even restrictions on wiring or geometric shapes might have some influence on the way how the control software is being built. This concerns, in particular, timing aspects which must be specified in the model as well, to be able to analyse the system model appropriately and to fully automatically generate code from the model specification. As a consequence a general UML-like language for all sorts of mechatronic systems in various domains such as e.g. transportation, production, or telecommunication might not exist but rather, if at all, a lot of domain specific adaptations and extensions of a possibly very general core language.

Finally, the definition of a precise (dynamic) semantics of such a language represents another major challenge. The current approach which is to attach code pieces to particular model elements like e.g. states or transitions in a state chart or Petri Net, when it comes down to the detailed definition of fine-grained activities or functions, is not appropriate and makes further model analysis almost impossible. Consequently, also the definition of such fine-grained activities must be definable on a more abstract level than code.
The trend towards adaptibility and self-organisation, indicated by the Cognitive Operator level in the OCM hierarchy requires further sophisticated concepts as part of a modelling language (a discussion of some of these issues can also be found in [20], in particular concerning models at runtime supporting adaptability). A model must include the precise definition of learning and improvement which basically corresponds to some kind of rule-based system expressing the permitted changes of component structure and behaviour. However, it is important to specify precisely the limits of such changes, because those changes will happen during the operation of the system and consequently should be safe. In a little more detail, one could imagine the CO developing a new plan and corresponding behaviour for a particular system component. The change initiated by the CO on the RO might require also a reconfiguration of the underlying controller structure and this might violate particular timing constraints which have been specified before. Then the change must be rejected and maybe modified by the CO. Checking those constraints vs. a proposal for change made by the CO is a very important part of keeping the system safe during runtime.

As a very first step in this direction, the following example taken from the RailCab project illustrates this idea. The realtime state chart of Fig. 6 represents a (small) part of the behavioural specification of a shuttle component. A change of this behaviour could be formally and graphically expressed by a graph transformation rule.

Such a rule could specify the necessary change of the automaton of Fig. 6, when the planning component of a shuttle, namely the CO, identifies a possible improvement of the shuttle behaviour which should be introduced during operation of the system. Such an improvement could be based, for example, on detecting that the protocol defined by the automaton in Fig. 6 yields a lot of network traffic, i.e. each shuttle in a convoy has to communicate with his two neighbours and, in addition, with the convoy leader (not specified in the example here). An improvement to cut down on network traffic could be to identify not only a convoy leader and the neighbours as the contact points for all other shuttles but a single reference contact point in a convoy, which takes care of all other necessary communication with the other reference contact points in the convoy. This requires the extension of the shuttle behaviour definition by a further state (called convoyMiddle here) and the corresponding state transitions. (Of course the corresponding protocol definitions have to be extended as well. As a sketch of an idea, the formal definition of such a change defined by a graph transformation rule and the result of applying this rule to the state chart defining the shuttle behaviour, is given in Fig. 7.

In general, such changes could include quite complex operations not only on an automaton (or Petrinet) or the like but also on the corresponding component structure. Component structure changes might be e.g. the creation or deletion of a port or connection or a change in the hierarchy.

The advantage of using graph transformation rules is twofold. They are a suitable visual formalism to express changes of graph-like structures which are the basis of all visual specification formalisms. In addition, they have a well-defined formal underpinning which enables to define a formal and precise semantic definition of rule-based systems given by graph transformations.

**Code generation.** Target platform specific code generators do not provide for enough flexibility to adjust code generation to often restricted but changing available hardware resources and requirements. If the code generator or rather its underlying algorithm knew about the changing constraints and the change in the model, it could generate optimized code based on currently available resources and currently needed pieces of the control software. This especially concerns the generation of code which considers all realtime constraints specified in the model. The optimized code could replace the currently existing code, if the generator has been verified (see below). This would however require that a code generator is flexible enough to be adjusted to and to even influence the underlying platform. In principle, this is possible by using Field Programmable
Gate Arrays which allow for a flexible configuration of the hardware resources, i.e. available processors and memory.

**Analysis.** From the point of view of analysis, or more specifically model checking, advanced mechatronic systems are mainly characterised by the fact that their state space is - due to the continuous parts - even larger than that of ordinary software systems (often already too large for model checking) and sometimes also infinite. This makes verification which could achieve a formal proof of correct functioning hard. On the other hand, their deployment in safety critical areas imposes the highest degree of reliability on mechatronic systems. This makes testing alone which remains feasible for these systems unsuitable (besides being time and cost intensive). As a consequence a third way in between has to be found which incorporates a number of known analysis techniques, develops new ones and combines them all into a reliable yet practicable method. We discuss this along the lines of present and future mechatronic systems.

The integration of mechanical, electronical and software parts poses challenges which so far have only partly been addressed. For the analysis of today’s mechatronic systems we can identify the following shortcomings:

**Precise hybrid modelling** No hybrid modelling techniques exist today which are able to describe the diverse parts of a mechatronic system in a uniform and - what is indispensable for analysis - precise way. Current formalisms try to simply combine some of the existing modelling language from the three areas but most often without giving a meaning to this mixed use of diagrams.

**Integrated hybrid analysis** The three disciplines involved in the construction of mechatronic systems all have analysis techniques on their own. Instead of applying these in isolation, an integrated analysis framework is needed in which a particular type of analysis in one area supports/relies on/triggers an analysis in another area.

**Verification** Systems with discrete and continuous parts are intrinsically difficult to verify. Model checking of hybrid systems and the transfer of known verification techniques to the domain of hybrid systems remains a challenge.

A possible line of approaching these challenges might lie in exploiting knowledge about the specific domain. Mechatronic systems have particular characteristics (e.g. with respect to their structuring, the involved controllers) which might be utilized in the analysis. The structure, viz. specific combinations of discrete and continuous parts, might form the basis of compositional analysis techniques for mechatronic systems. Thus, a compositional approach could integrate analysis result achieved by different techniques, in particular could combine results obtained by mechanical and electrical engineers with those of software engineering. The engineer’s longstanding experience in constructing reliable systems and their specific knowledge about controllers furthermore makes it worthwhile to pursue approximation techniques that completely abstract from the continuous parts. Finally, current research developments for verification tightly coupling different techniques will broaden the class of hybrid systems manageable by automatic model checking.

Next generations mechatronic system will however come with new characteristics which make analysis even harder. In the future, mechatronic systems will consists of several "small" autonomous mechatronic agents, interacting with each other and adapting to changes in their environment. Adaptation according to measurements by an agent’s own sensors will be complemented by adaptations due to information obtained from other agents. The RailCab project already shows this new level of interaction: when driving in a convoy the shuttles exchange information about their positions and speed as to optimise the distance between them and thus power consumption. For analysis, this opens yet another dimension of complexity.

**Volatility** Evolution according to new data from the environment will be one main characteristic of future advanced mechatronic systems. The behaviour of such systems will thus not be completely fixed during design, but is allowed to adapt to environmental changes. The permitted degree of change might partially be laid down by model transformations being part of the model itself. Verification thus has to show that the system remains safe under all possible influences from the environment.

A possible way of tackling volatility in the analysis are methods reasoning on the changes themselves. Currently, such techniques take the following approach: correctness properties are proven for the initial states (of the system) and in addition shown to be preserved under transitions of the state. Examples of this are inductive verification techniques like [14, 28, 7] or methods for change analysis and property preservation for inheritance [32, 33]. For volatile mechatronic systems, such methods need to be lifted to transformations of the model itself. A prerequisite – like for all analysis techniques – is again a precise description of model transformations. Safety properties then have to be shown to be preserved under all possible transformations.

An example illustrating this idea is again taken from the RailCab project. Consider the definition of model changes...
defined by a system of graph transformation rules. Such a system of course might define an infinite state space of possible changes to a given model. The problem is now to check whether none of these changes produces a model which violates certain constraints. Standard techniques which are based on exploring all reachable states obviously must fail. Here the approach is to define upfront which models or rather small cutouts of models expressed by a graph-like description represent violations of constraints. Then an inductive check takes a backward approach and tries to verify whether a backward application of a rule produces a correct model from an incorrect one, i.e. one which violates certain constraints. If this is the case, one can deduce that the rule could produce an incorrect model if applied in forward mode. This strategy might lead to detecting false positives, i.e. rules which actually do not produce incorrect models but it can still be used as an indication to the modeller or warning to an engineer that something might go wrong.

As an example, take the change sketched above which introduces the role of a convoyLeader and thereby changes the behavioural definition of a shuttle component. A constraint to be checked could be that the extension to the specification, namely the introduction of the new state and its corresponding transitions, does not violate existing timing constraints of the transitions of the existing model.

Finally, in the future this trend will brings us to self-organising mechatronic systems. These will partially be inspired by biological systems, where agents have the ability to reflect upon their own behaviour and to learn from others. Agents will act on their own behalf, trying to optimise their own benefits. This objective of local optimisation might necessitate forming coalitions with as well as competing for limited resources against other agents. In such a scenario, analysis has to cope with yet another phenomenon.

**Emergence** In self-organising systems new, unforeseen properties can emerge. For the analysis this means that the state space, which normally is the basis for verification, is unknown or not available for inspection. Analysis can only rely on incomplete, most likely local, knowledge about the system. The influence from the environment of an agent is unpredictable.

It is unclear today how an analysis framework for such advanced systems might look like. Most likely, it comprises all the before mentioned techniques which are combined in an integrated framework. Furthermore, as the aspect of self-organisation particularly introduces interaction with initially unknown components, the use of games in verification could be a promising approach.

As the analysis only takes place on the level of the model, safety of the realisation can moreover only be guaranteed when additional constraints are satisfied:

- Code generation has to be provably correct. Here, recent progress in research on verified compilers might prove useful, as well as current work on certified code generations [22].
- Assumptions on the physical systems made in the model have to be guaranteed in the realisation. This does for instance concern communication times for network connections or worst case execution times of operations. In self-organising systems with a large amount of data exchanges between components, security might furthermore become an issue.

It is in particular the latter point where an involvement of computer science disciplines other than software engineering is needed.

### 5 Conclusion

In this article, we have sketched current and future trends in the development of mechatronic systems. In particular, we have discussed the challenges involved in the construction of future advanced systems. Summarising, these can be roughly divided into two categories: the challenges arising from the collaboration of several different disciplines (which is already an issue today), and those due to the aspect of self-coordination which seems to be a main characteristic distinguishing current from future mechatronic systems. These are challenges to all involved disciplines, but in particular to software engineering. Key to a success in mastering them is the joint effort and collaboration of disciplines, within computer science as well as between computer science and engineering.

### References


