Symbolic Invariant Verification for Systems with Dynamic Structural Adaptation [*]

Basil Becker[1] Dirk Beyer[2] Holger Giese[1] Florian Klein[1] Daniela Schilling[1]

[1] Software Engineering Group
University of Paderborn, Germany
{basilb,hg,fklein,das}@upb.de

[2] Models and Theory of Computation
EPFL, Lausanne, Switzerland
Dirk.Beyer@epfl.ch

ABSTRACT

The next generation of networked mechatronic systems will be characterized by complex coordination and structural adaptation at run-time. Crucial safety properties have to be guaranteed for all potential structural configurations. Testing cannot provide safety guarantees, while current model checking and theorem proving techniques do not scale for such systems. We present a verification technique for arbitrarily large multi-agent systems from the mechatronic domain, featuring complex coordination and structural adaptation. We overcome the limitations of existing techniques by exploiting the local character of structural safety properties. The system state is modeled as a graph, system transitions are modeled as rule applications in a graph transformation system, and safety properties of the system are encoded as inductive invariants (permitting the verification of infinite state systems). We developed a symbolic verification procedure that allows us to perform the computation on an efficient BDD-based graph manipulation engine, and we report performance results for several examples.

Categories and Subject Descriptors: D.2.4 [Software Engineering]: Software/Program Verification — *Formal methods*; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs — *Invariants*; D.2.11 [Software Engineering]: Software Architectures — *Domain-specific architectures*; D.2.2 [Software Engineering]: Design Tools and Techniques — *Computer-aided software engineering (CASE)*;

General Terms: Design, Reliability, Verification

Keywords: Formal verification, Graph transformation systems, Structural invariants, Transition invariants, Symbolic algorithms, Embedded systems, Mechatronics

[*]This research was supported in part by the SFB 614 of the DFG, the MICS NCCR of the SNSF, and the International Graduate School of Dynamic Intelligent Systems of the University of Paderborn.

1. INTRODUCTION

Mechatronic systems [6] combine traditional mechanical and electrical engineering with technologies from software engineering in order to provide reliable technical solutions for complex real-world problems. In the future, advanced mechatronic multi-agent systems are expected to exhibit increasingly complex, context-dependent behavior. The key to enhance their behavior and pushing the limits of feasible functionality is the agents' ability to cooperate in dynamic local and global networks, using complex real-time coordination and structural adaptation. The envisoned systems will not only exhibit self-adaptive behavior (cf. [19, 23]) within individual agents, but also at the inter-agent level.

As mechatronic systems are usually safety-critical, it has to be ensured that any configuration that is reachable at run-time is safe w.r.t. a given set of safety properties. Consequently, the development of such systems has to include rigorous verification activities that can guarantee the identified crucial safety properties even for those real-time coordination features that result from the structural adaptation at run-time. Techniques such as testing the system in several test environments and in its operation environment are not sufficient for this task due to their incomplete nature. On the other hand, complete automated verification techniques do not scale: current model checking approaches can prove safety properties for models of moderate size only, and semi-automatic approaches such as theorem proving require advanced proof skills, which are usually not available.

Previous verification approaches for systems with structural adaptation were based on exhaustive state exploration; they are limited to finite state models of rather small size [3, 14, 16, 20, 21, 24]. It is even more problematic that they require a *known* initial system topology, whereas in practice, the system design has to support a huge set of well-formed initial configurations. The structural adaptation could, in principle, lead to an unbounded number of reachable system configurations, which was not considered so far.

We present a verification technique that can handle the real-time coordination and structural adaptation of arbitrarily large multi-agent systems from the mechatronic domain. We model the system's ontology using UML class diagrams, the system's states as graphs —represented by UML object diagrams, which provide a suitable, user-friendly graphical notation for graphs—, and the system's behavior as a set of graph transformation rules —represented by story patterns, which are an extension of object diagrams for modeling behavior. Safety requirements of the system are modeled using graph patterns: for example, the set of unsafe system states is represented by a set of graph patterns, which we call 'forbidden graph patterns'.

The question answered by traditional verification is whether an unsafe system state is reachable from an initial system state. In other words: can one of the initial graphs be transformed into one of the forbidden graphs using a sequence of graph transformations from the set of transformation rules defining the system's behavior? In our application domain, we neither know all initial states nor can we compute *all* reachable states of the system. However, we can inspect every transformation rule to find out whether it can transform a safe graph into an unsafe graph. Since we cannot consider the —usually infinite— complete set of safe graphs, we try the opposite: we verify that the backward application of a rule to a forbidden graph pattern cannot lead to a graph pattern that represents a safe system state. Once we have verified this property for every forbidden graph pattern (there is a finite number of them, although they can represent infinitely many states) and for every transformation rule of the system (once again a finite set), we have proven that the system can never enter an unsafe state.

This technique is well-known as the verification of transition invariants, and many approaches to make it practical exist (e.g. [5]), but transition invariants have not yet been applied to systems with structural adaptation. Applied to the area of software and hardware verification, this approach is limited due to the overapproximation that prevents the method from proving many interesting properties. However, modeling the system transitions of mechatronic systems with structural adaptation as graph transformations allows us to capture the most interesting part of the system in the transformation rules. We report on a number of examples based on the R&D project RailCab (http://www.railcab.de) in the modeling section and in the evaluation section.

We provide a tool implementation that scales to large systems, and developed —besides the traditional explicit algorithm, which is based on traditional data structures for graph manipulation— a symbolic algorithm for our approach. The symbolic encoding of the problem allows us to run the algorithm on engines that are optimized for symbolic computations (e.g., BDD engines and SAT solvers). We integrated both the explicit and the symbolic algorithm into the UML CASE tool Fujaba (http://www.fujaba.de). We used the GROOVE engine [21] for comparing our results to a model-checking approach, i.e., computing all reachable graphs from a given initial graph. The explicit algorithm is completely implemented within the Fujaba framework. For the symbolic encoding, we use the relational programming language RML and the BDD-based calculator CrocoPat [4]. Our experiments show that model-checking and the explicit approach work well for small examples, while the symbolic approach has the potential to scale to larger graph patterns.

The paper is structured as follows: The modeling approach is introduced by means of a running example in Sect. 2. Section 3 defines the underlying formal model in terms of graphs and graph transformation systems. Our new approach for the invariant verification of structural properties is explained in Sect. 4, where we also discuss the explicit and symbolic algorithms for the verification. We present the results of several performance experiments in Sect. 5, and Sect. 6 relates our work to other approaches.

2. MODELING APPROACH

2.1 Advanced Mechatronic Systems

Example. As an application example that is representative of both the challenges and the potential of advanced mechatronic systems, we use a system of autonomous shuttles navigating a railway network. Inspired by the R&D project RailCab, the intent is providing safe, energy-efficient individual transportation.

To this end, shuttles have the ability to minimize drag by forming contact-free convoys. However, increased efficiency comes at the cost of reduced safety margins. In order to avoid collisions, shuttles need to precisely coordinate their spacing and anticipate decelerations. This can only be achieved cooperatively, as distances cannot reliably be measured in bends or at junctions. A wireless network allows shuttles to communicate speeds, positions, and intended driving moves. Besides, the convoy mode constrains the admissible behavior, e.g., reduces the maximum speed and the brake force of leading shuttles the more tightly-spaced a convoy is.

To ensure safe operation, the required *real-time coordination* thus needs to be established reliably and satisfy certain safety requirements (e.g., consistent convoy mode, no deadlocks). In case of a complete communication failure, all shuttles need to perform a controlled emergency braking maneuver, resulting in a trivially safe system state.

Approach. From the information processing perspective, active mechatronic subsystems of such a system can be seen as autonomous agents, enhancing their behavior through cooperation and the exploitation of contextual knowledge. Their ability to establish and relinquish real-time coordination as required is the key to their flexibility, as it allows changing the system structure at run-time. Such structural adaptation is a powerful concept that enables more sophisticated solutions, including complex software-based coordination and information management.

Our modeling approach uses a formal model of the system's ontology as its foundation. This model includes the underlying structure, assumed structural constraints, hazards, and a coarse-grained abstraction of the underlying physical behavior. On top of this, we layer a control architecture consisting of various social structures and coordination pattern types that interrelate the agents and exclude the specified hazards.

When verifying that the structural evolution and the instantiation rules of the control architecture are sufficient w.r.t. the safety requirements, we rely on two restricting assumptions that are plausible in a mechatronic context: (1) Coordination patterns refer to a local context and only contain a limited number of participating agents, which bounds the number of elements we need to consider. (2) Required reaction times for establishing a coordination pattern are an order of magnitude larger than the required reaction times within the pattern itself, which enables us to verify relevant structural properties without explicitly considering real-time aspects as the agent will always be able to react within this generous time frame.

2.2 Modeling

All our notations are based on the UML. Beside the standard notations, we use MECHATRONIC UML [7, 11, 13] for the specification of real-time coordination patterns. We also employ story patterns, an extension of UML object diagrams based on the theory of graph transformation systems (cf. [18]), for expressing structural changes and properties.

Physical System. We model the *ontology* of the system using UML class diagrams. Figure 1 specifies the physical entities of the system —shuttle agents and tracks— and their relationships. Tracks are short segments with room for a single Shuttle. The successor association connects them into a network. A Shuttle's position is expressed by the on association; the go association encodes physical movement towards a Track. Note that, even though we are not using attributes and are thus abstracting from the Shuttle's actual position on the Track, this level of abstraction is sufficient for designing the structural adaptation of the real-time coordination.[1]

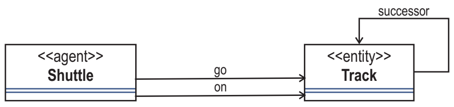


Figure 1: Class diagram modeling physical entities

Concrete states instantiating the ontology can be modeled using UML object diagrams. We use this to specify unsafe conditions, i.e., states that must not occur during the execution of the system. Specifically, this allows expressing *hazards* and *accidents*.

All *hazards* and *accidents* a system is supposed to exclude need to be defined explicitly. In our example, we restrict our attention to shuttle collisions, characterized by two shuttles sharing a track (see Fig. 2).

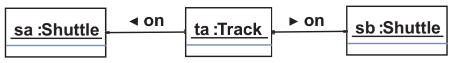


Figure 2: Diagram specifying a Collision accident

Note that cardinalities in conditions are currently not directly supported by our approach. It is possible, however, to encode cardinalities that are critical for the system's correctness by means of additional conditions (e.g., forbidding Shuttles with two on associations).

Story patterns allow modeling the behavior of the system. Story patterns basically consist of two object diagrams specifying concrete states, a precondition and a postcondition. If the precondition is matched, i.e., occurs in a state, that occurrence is transformed to correspond to the postcondition. By extending object diagrams with appropriate stereotypes, the pre- and postcondition can be compactly specified as a single object graph where unmarked elements remain constant, elements annotated with ≪destroy≫ are erased and elements annotated with ≪create≫ are created. Crossed out elements are parts of the precondition that must *not* occur in a state; otherwise the pattern is not applicable.

[1]Using attributes is possible even though in this paper all properties are expressed in terms of structural relationships between objects. There is no categorical limitation to that effect in either story patterns or our approach.

We use story patterns for expressing all kinds of state changes, e.g., agent behavior, rules, and physical processes. The story patterns in Fig. 3 and 4 describe the physical process of a Shuttle moving from one Track to the next. As we strive for an adequate description of the physical level that does not abstract away relevant problems, the specified behavioral rules do in fact allow a collision to happen.

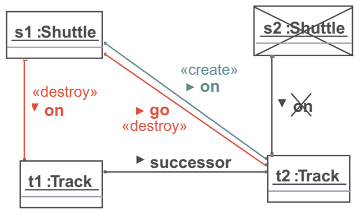


Figure 3: moveSingle: Story Pattern for moving a shuttle to an empty track.

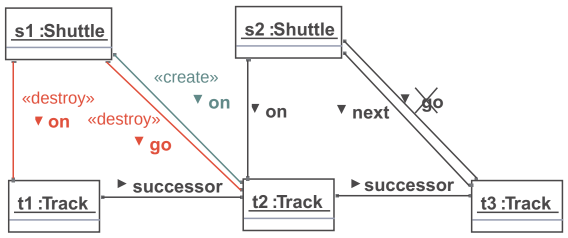


Figure 4: moveMultiple: Story Pattern for crashing a shuttle into another, immobile shuttle.

Coordination Architecture. In order to achieve the desired properties, we now extend the specification with social structures controlling the architectural evolution and the real-time coordination, prominently using MECHATRONIC UML coordination patterns [13]. As a means of structuring the problem domain, we define a hierarchy of *cultures* [17], each of which is responsible for ensuring a set of specific system properties.

A culture is a set of subcultures, roles, instantiation rules, behavioral rules, professed intentions, and invariants. A *community* is a dynamically formed group of agents implementing a specific culture. *Instantiation rules* are responsible for creating and destroying communities and assigning roles to agents, i.e., the actual structural adaptation. *Behavioral rules* specify valid physical and social agent behavior. They may also provide a social interpretation of the agents' actions. *Professed Intentions* are such interpretations of an agent's intentions, e.g. as a commitment to a specific course of action. *Invariants* encode constraints and properties guaranteed by the culture. All agents belong to a global default community that can be used for bootstrapping.

Coordination patterns can be seen as a restricted type of culture without subcultures and proprietary professed intentions, i.e., pattern instantiation can be subsumed under community instantiation. In the case of MECHATRONIC UML coordination patterns, roles correspond to communication protocols and are linked by connectors representing communication channels. Invariants serve to express constraints for each role and the overall pattern.

The central idea of the approach is to start with solutions for small, specific problems, and then compose them into the overall multi-agent system. Agents are seen as components that implement pattern roles and internally reconcile potential conflicts. The corresponding modeling process consists of five steps: (1) After specifying a coordination pattern / culture, (2) formal verification is used to ensure that it conforms to its invariants. (3) The pattern is then stored for reuse (4) and used to create components by refining roles to ports and properly synchronizing them. (5) Refinement and synchronization again need to be verified against the role constraints. Any syntactically correct composition of verified components is then guaranteed to yield a correct system without any additional verification (cf. [13]).

Shuttle Culture. At the specification level, the physical ontology (Fig. 1) is extended with conceptual elements to yield the augmented ontology presented in Fig. 5. The (implied) ShuttleCulture introduces the next association, which represents a commitment (marked by the stereotype) to go to a specific Track the next time a go-Rule is executed. Besides, it contains the DistanceCoordinationCulture as a subculture. The DistanceCoordinationPattern, which instantiates that culture, groups two Shuttles which take on the rear respectively front *role*, again marked by stereotypes.

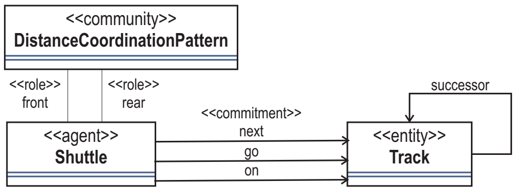


Figure 5: Augmented class diagram

The ShuttleCulture introduces two behavioral rules: goSimple1 (see Fig. 6) allowing a solitary (i.e., not following another Shuttle) Shuttle to move freely where no Tracks join, and goSimple2, forcing a solitary Shuttle to give way at a switch where Tracks join. These rules imply the convention that Shuttles respect the commitment expressed by their next association.

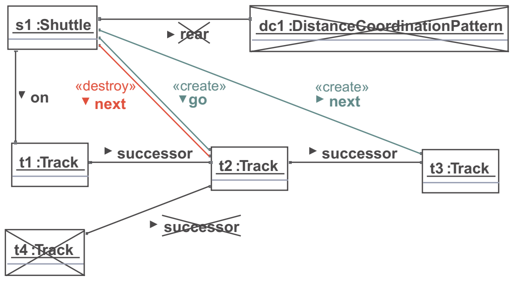
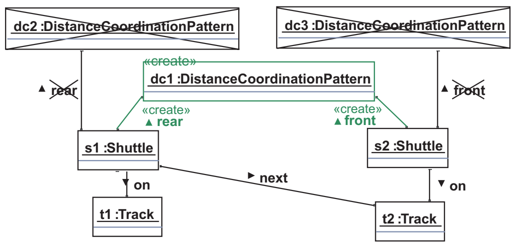


Figure 6: Behavioral rule: unrestricted movement for a solitary Shuttle

Distance Coordination Culture. The actual collision avoidance is realized by the ShuttleCulture's subculture responsible for distance coordination. The culture achieves this by instantiating a DistanceCoordinationPattern —thereby disabling the goSimple rules and enabling the provided behavioral rules for coordinated movement— once a Shuttle approaches another.

The instantiation rule createDC creates a DistanceCoordinationPattern, and thus a new community, if there is a hitherto unconnected Shuttle on a Shuttle's next Track (see Fig. 7). The rule deleteDC removes the pattern as soon as the rear Shuttle no longer has a go or next association to the front Shuttle's location, i.e., the Shuttles have moved away from each other.



dc2 :DistanceCoordinationPattern
dc3 :DistanceCoordinationPattern
«create»
dc1 :DistanceCoordinationPattern
rear
front
«create»
rear
«create»
front
s1 :Shuttle
s2 :Shuttle
on
next
on
t1 :Track
t2 :Track

Figure 7: Instantiation rule: creating a DistanceCoordinationPattern

The behavioral rules goDC1 (see Fig. 8) and goDC2 only allow the rear Shuttle to move, i.e., go, once the front Shuttle has decided to move. This prevents the moveMultiple rule (for crashing into a stationary Shuttle) from ever applying, thus in turn preventing collisions.

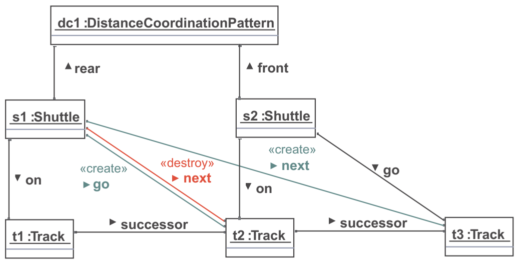


Figure 8: Behavioral rule: Coordinated movement

An invariant that is implied in this specification is that a Shuttle will never try to go to a Track occupied by another Shuttle without coordinating its movement, i.e., making sure the other Shuttle is moving, which would constitute a hazard. Though not required for the operational correctness of the model, this implied condition (see Fig. 9) needs to be made explicit, along with several structural constraints restricting cardinalities, in order for the specification to pass the inductive invariant checking introduced below.

In the remainder of the paper, we will outline the semantic underpinning of the employed concepts and our approach to automatically verify that collisions are effectively avoided.

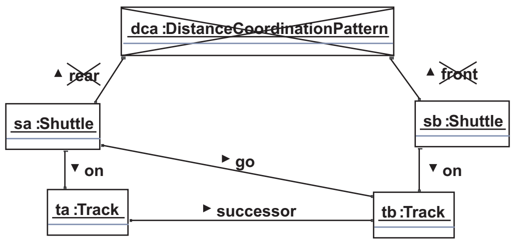


Figure 9: Invariant: No uncoordinated movement of Shuttles in close proximity, which would constitute a hazard

3. FORMAL MODEL

In the preceding section, we used class diagrams and story patterns to define possible system states and possible system transitions. This section defines the formal model of graph transformation systems in general, and story patterns in particular, which we use in our approach for the verification of safety properties.

Graphs. A system state, given as an object diagram, can be encoded as a graph by modeling objects as nodes and links as edges. The system model is based on a set $\mathcal{N}$ of nodes, a set $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ of edges, and a set $\mathcal{T}$ of types. The type of a node or edge is defined by the relation $T : (\mathcal{N} \cup \mathcal{E}) \times \mathcal{T}$, i.e., x is of type t if $(x, t) \in T$. A *graph* $G = (N, E)$ consists of a set $N \subseteq \mathcal{N}$ of nodes and a set $E \subseteq N \times N$ of edges.

Let $G_1 = (N_1, E_1)$ and $G_2 = (N_2, E_2)$ be two graphs. The union $G_1 \cup G_2$ is defined as $(N_1 \cup N_2, E_1 \cup E_2)$, the intersection $G_1 \cap G_2$ as $(N_1 \cap N_2, E_1 \cap E_2)$, and the subtraction $G_1 \setminus G_2$ as (N, E) with $N = N_1 \setminus N_2$ and $E = (E_1 \setminus E_2) \cap (N \times N)$.

Example 1. **(Graph)** Consider the set $\mathcal{N} = \{sa, sb, ta\}$ of nodes, the set $\mathcal{E} = \mathcal{N} \times \mathcal{N}$ of edges, the set $\mathcal{T} = \{\mathit{Shuttle}, \mathit{Track}, \mathit{on}\}$ of types, and the type relation T with $T = \{(sa, \mathit{Shuttle}), (sb, \mathit{Shuttle}), (ta, \mathit{Track}), ((ta, sa), \mathit{on}), ((ta, sb), \mathit{on})\}$. Then $G = (\mathcal{N}, \{(ta, sa), (ta, sb)\})$ is the graph representing the collision accident depicted in Fig. 2.

Graph Patterns. A *graph pattern* $P = (N^+, N^-, E^+, E^-)$ consists of two sets N^+ and N^- of positive and negative nodes, and two sets E^+ and E^- of positive and negative edges. The sets E^+ and E^- are subsets of $(N^+ \cup N^-) \times (N^+ \cup N^-)$. We further use P^+ to denote P restricted to N^+ and E^+. A graph pattern represents the set of graphs that match the pattern. A graph G *matches* a graph pattern P if there exists an isomorphic function m that maps the positive nodes and positive edges of P to nodes and edges of G, respectively, and m cannot be extended in such a way that it matches any negative node or negative edge of P to a node or edge in G, respectively. The matching function m preserves types, i.e., a node or edge may only be mapped to a node or edge of the same type, respectively. There can be an arbitrary number of negative elements, but negative nodes have to be connected with at least one positive node and may not be connected with each other.

A graph pattern P *matches* a graph pattern P' if there exists an isomorphic function iso that maps all positive elements of P to positive elements of P' and all negative elements of P to negative elements of P'. The set of all such isomorphic functions is denoted by ISO. If P matches P', we say that P is a subpattern of P', and write $P \sqsubseteq P'$.

Graph patterns that are used to describe system properties can be divided into required and forbidden patterns. A *required* pattern must always be fulfilled during system execution (system invariant), whereas a *forbidden* pattern must never be fulfilled (system hazard, accident).

Graph Transformation Systems. A graph transformation transforms one graph into another one by creating new graph elements (nodes or edges) and/or removing existing graph elements. Transformation rules define sets of possible graph transformations. If a graph represents the state of a system, a graph transformation represents an update of the system's state, and a sequence of transformations represents an execution of the system.

Story patterns are extended graph patterns that allow the annotation of graph elements with the stereotypes ≪create≫ and ≪destroy≫. As introduced informally above, story patterns are used to specify transformation rules. A graph transformation rule $(L, R)_r$ consists of two graph patterns, a left hand side L (LHS) and a right hand side R (RHS). L consists of those elements of the story pattern that are not annotated with ≪create≫, including negative elements, whereas R consists of all elements not annotated with ≪destroy≫. The elements annotated with ≪create≫ will be created by the rule (elements from $R^+ \setminus L^+$), while those annotated with ≪destroy≫ will be deleted (elements from $L^+ \setminus R^+$). Elements without annotations are preserved by the application (elements from $L^+ \cap R^+$).

A *graph transformation system* (GTS) $S = (\mathcal{R}, \mathit{prio})$ consists of a set of graph transformation rules $\mathcal{R}$ (defined by a set of story patterns), defining all possible transformations in the transformation system, and a priority function $\mathit{prio} : \mathcal{R} \to \mathbb{N}$, which assigns a priority to each rule (the higher the number assigned to a rule the higher the rule's precedence). An additional set of initial graphs may describe the initial states of the system.

A rule $(L, R)_r$ is *applicable* to a graph G if G matches L *and* G does not match the LHS L' of any other rule $(L', R')_{r'}$ with higher priority. During the *application* of a rule $(L, R)_r$ to a graph G, the elements that are in L^+ but not in R^+ are removed from G, and elements that are in R^+ but not in L^+ are added to G.[2] We write $G \to_r G'$ if rule r can be applied to graph G and the application results in graph G'. We write $G \to^* G'$ if G is transformed into G' by a (possibly empty) sequence of rule applications. Given a graph transformation system S and a graph G, the set of graphs producible (i.e., the set of states reachable) by applying rules from S to G is denoted by $REACH(S, G) = \{G' \mid G \to^* G'\}$.

We extend the notion of rule application to graph patterns in a natural way. The application of a rule to a graph pattern that represents a set of start graphs results in a graph pattern that represents the set of all graphs resulting from applying the rule to a start graph. The backward application of a rule $(L, R)_r$ to a graph pattern P is defined as $\mathsf{ApplyBack}(r, P) := P \setminus (R \setminus L) \cup L$ if $R \sqsubseteq P$ and $(L \setminus R) \cap P = G_\emptyset$ holds, with $G_\emptyset = (\emptyset, \emptyset)$ is the empty graph.

4. VERIFICATION

This section states the verification problem and proposes to solve the original problem by checking inductive invariants, which works well in practice for systems from the considered mechatronic domain. We want to verify that none of the structural adaptations within the system —formally described by transformation rules— leads to an unsafe system state. We introduce an explicit and a symbolic implementation of the procedure. The goal of the symbolic implementation is to increase the efficiency by using a relational calculator that is optimized for the required operations.

[2]Our notion of application is a restricted version of the *Single Pushout* approach (cf. [22]). The approach is restricted in such a way that the identification condition as well as the dangling edge condition are always fulfilled. This restriction further ensures that the backward application of rules is always possible when an automatic extension procedure adds the corresponding tests for all created resp. removed elements in form of negative elements to L resp. R.

Verification Problem. Safety-violation conditions (hazards, accidents) are represented by a set of forbidden graph patterns $\mathcal{F} = \{F_1, \dots, F_n\}$. The graph G fulfills the safety property $\Phi_\mathcal{F}$, denoted by $G \models \Phi_\mathcal{F}$, if G matches none of the graph patterns in $\mathcal{F}$. If G matches a forbidden graph pattern $F \in \mathcal{F}$, we call G a *witness* for the property $\neg\Phi_\mathcal{F}$. The property $\Phi_\mathcal{F}$ is an *operational invariant* of the GTS S iff $G \models \Phi_\mathcal{F}$ for all $G \in \mathsf{REACH}(S, G^0)$, for a given initial graph G^0 (cf. [9]).

In general, operational invariants cannot be checked fully automatically, because graph transformation systems with types are Turing-complete. Therefore, it is impossible to provide complete automatic verification procedures. Model checking approaches for GTS can only be employed if G^0 and $\mathsf{REACH}(S, G^0)$ are finite, or finitely representable.

The property $\Phi_\mathcal{F}$ is an *inductive invariant* of a GTS $S = (\mathcal{R}, prio)$ if for all graphs G and for all rules $r \in \mathcal{R}$ the following holds: $G \models \Phi_\mathcal{F} \wedge G \rightarrow_r G'$ implies $G' \models \Phi_\mathcal{F}$. If this is the case and the initial graph G^0 fulfills the property, then $\Phi_\mathcal{F}$ is also an *operational invariant* (the reverse implication does not hold in general).

As, at design-time, we cannot place undue restrictions on the system's deployment, the set of initial states is not known, in the considered domain. In addition, the reachable state space $\mathsf{REACH}(S, G^0)$ is usually infinite. Therefore, we propose to prove inductive invariants of the GTS to verify the system's safety. In the next subsection, we present our verification approach which covers infinite state systems and allows arbitrary correct initial deployments.

Checking Inductive Invariants. In order to solve this verification problem, we reformulate the definition of an *inductive invariant* in a more readily falsifiable form: a property $\Phi_\mathcal{F}$ is an inductive invariant of a GTS $S = (\mathcal{R}, prio)$ if and only if there exists no pair (G, r) of a graph G and a rule $r \in \mathcal{R}$ such that $G \models \Phi_\mathcal{F}$, $G \rightarrow_r G'$ and $G' \not\models \Phi_\mathcal{F}$. We call such a pair (G, r) a *counterexample*, as it witnesses the violation of property $\Phi_\mathcal{F}$ by rule r.

To verify whether a counterexample exists, we can exploit the fact that the application of a rule can only have a local effect: only that clearly delimited part of a graph that is matched by the rule can be affected by its application. When a counterexample (G, r) occurs, the local modification of G by rule r is necessarily responsible for transforming the correct graph G into a graph that violates the property.

A forbidden graph F can result from a rule application in two ways. Firstly, the rule $(L, R)_r$ may complete the positive part of a forbidden graph with an element from $R^+ \setminus L^+ \cap F^+$. Secondly, the rule may delete elements of the start graph that were negative elements of a forbidden graph pattern F and thus prevented it from matching, i.e. elements from $L^+ \setminus R^+ \cap F^-$. Our restricted notion of application requires these latter elements to be connected to an element from $L^+ \cap R^+ \cap F^+$ to avoid dangling edges. As any possible counterexample must fall into either one or both of these categories, G' must therefore contain an intersection between the positive nodes of the RHS of the rule and the forbidden graph, i.e., $R^+ \cap F^+$.

Consequently, we can check that no counterexample exists (and $\Phi_\mathcal{F}$ is thus an inductive invariant) by considering the finite set Θ of graph patterns P' that are combinations of a RHS R of a rule r and a forbidden graph pattern $F \in \mathcal{F}$: $\Theta(F, R) = \{iso(F) \cup R \mid iso \in ISO \text{ and } iso(F) \cap R \neq \emptyset\}$. For each of these combined graph patterns $P' \in \Theta(F, R)$ we only need to check if there exists a forbidden graph pattern

Algorithm 1 $\mathsf{SearchCounterexample}(r, F, \mathcal{C})$

```
Input: Rule $(L,R)_r$, GraphPattern $F$, Set⟨GraphPattern⟩ $\mathcal{C}$
Output: Set⟨Counterexample⟩
Variables: Set⟨GraphPattern⟩ $tgpSet$, $sgpSet$
1: $tgpSet := \Theta(F, R)$
2: $sgpSet := \{P \mid P' \in tgpSet \wedge P = \mathsf{ApplyBack}(r, P')\}$
3: return $\{(P, r) \mid P \in sgpSet \wedge \nexists F \in \mathcal{C} : F \sqsubseteq P\}$
```

Algorithm 2 $\mathsf{CheckInvariant}(S, \mathcal{F})$

```
Input: GTS $S = (\mathcal{R}, p)$, Set⟨GraphPattern⟩ $\mathcal{F}$
Output: Set⟨Counterexample⟩
Variables: Rule $r$, Set⟨GraphPattern⟩ $candSet$, GraphPattern $F$,
           Set⟨Counterexample⟩ $res$
 1: for all $(L,R)_r \in \mathcal{R}$ do
 2:   $candSet := \mathcal{F} \cup \{L' | (L', R')_{r'} \in \mathcal{R} \wedge prio(r') > prio(r)\}$
 3:   for all $F \in \mathcal{F}$ do
 4:     $res :=$ SearchCounterexample($r, F, candSet$)
 5:     if $res \neq \emptyset$ then
 6:       // Counterexample found.
 7:       return $res$
 8:     end if
 9:   end for
10: end for
11: return $\emptyset$
```

$F' \in \mathcal{F}$ with $F' \sqsubseteq P$ for the graph pattern P with $P \rightarrow_r P'$. Let $S = (\mathcal{R}, prio)$ be a GTS, P' be a graph pattern, and $r \in \mathcal{R}$ be a transformation rule of S. The pair (P, r) is a counterexample for $\Phi_\mathcal{F}$ if the following conditions hold:

1. $P' \in \Theta(F, R)$ for some $F \in \mathcal{F}$, and
2. $P \rightarrow_r P'$, i.e., the rule r can be applied to graph pattern P and the resulting graph pattern is P' (this implies that no $r' \in \mathcal{R} \setminus \{r\}$ exists with $prio(r') > prio(r)$ that matches G, due to the definition of rule application), and
3. there exists no $F' \in \mathcal{F}$ with $F' \sqsubseteq P$, i.e., P is safe.

We use the above conditions in our algorithms to check if a counterexample exists.[3] Algorithm 1 performs this check for a given rule $(L, R)_r \in \mathcal{R}$, a forbidden graph pattern $F \in \mathcal{F}$, and a set $\mathcal{C}$ of candidate graph patterns. The algorithm first computes the set of all possible target graph patterns ($tgpSet$) for R and the forbidden graph pattern F (using function Θ). The source graph patterns ($sgpSet$) are then determined using $\mathsf{ApplyBack}$. The set of candidate graph patterns $\mathcal{C}$ is the union of the set of forbidden graph patterns $\mathcal{F}$ and the left hand sides (LHS) of all rules with higher priority. If a graph pattern from $\mathcal{C}$ matches the source graph pattern, the application of r is either irrelevant, as the source graph pattern already represents a forbidden state, or impossible, because it is preempted by another matching rule with higher priority. Otherwise, if no graph pattern from $\mathcal{C}$ matches, the source graph pattern P represents graphs that can be transformed into unsafe graphs by applying r, and

[3]In the case of negative elements in the rules or graph patterns, the check is a sufficient but not a necessary criterion for a counterexample, while without negative elements the check is sufficient and necessary. This limitation exists due to rare cases in which no forbidden graph pattern matches the identified source graph pattern, due to missing negative elements, but no concrete counterexample graph that matches the source graph pattern but not some forbidden graph pattern can be constructed.

Algorithm 3 SearchCounterexampleExplicit$(r, F, \mathcal{C})$

```
Input: Rule $(L,R)_r$, GraphPattern $F$, Set⟨GraphPattern⟩ $\mathcal{C}$
Output: Set⟨Counterexample⟩
Variables: GraphPattern $target, source, cand,$
Variables: Boolean $enabled$, Set⟨GraphPattern⟩ $candSet$
 1: for all $target \in \Theta(F,R)$ do
 2:   $source :=$ ApplyBack$(r, target)$
 3:   $enabled := true$
 4:   $candSet := \mathcal{C}$
 5:   while $enabled \wedge candSet \neq \emptyset$ do
 6:     choose $cand \in candSet$
 7:     $candSet := candSet \setminus \{cand\}$
 8:     if $cand \sqsubseteq source$ then
 9:       $enabled := false$
10:     end if
11:   end while
12:   if $enabled$ then
13:     // Counterexample found.
14:     return $\{(source, r)\}$
15:   end if
16: end for
17: return $\emptyset$
```

the pair (P, r) is included in the resulting set of counterexamples.

Algorithm 2 checks for a given GTS $S = (\mathcal{R}, prio)$ and a set of forbidden graphs $\mathcal{F}$ whether $\Phi_{\mathcal{F}}$ is an inductive invariant of S. It enumerates all pairs of rules $r \in \mathcal{R}$ and forbidden graph patterns $F \in \mathcal{F}$ and applies Alg. 1 to them. For each pair, Alg. 1 either returns a non-empty set of counterexamples, or the empty set if no counterexample exists. The property $\Phi_{\mathcal{F}}$ is an inductive invariant if the algorithm does not find any counterexamples.

Explicit implementation. Algorithm 3 is an implementation of Alg. 1 based on an explicit graph representation and on explicit operations. It stops the search as soon as it encounters the first counterexample. A serious problem of the explicit algorithm is that the number of target graph patterns in the set $\Theta(F, R)$ grows exponentially w.r.t. the size of the involved graphs. If there is no counterexample, the for all loop in line 1 will have to iterate over each of the target patterns.

Symbolic implementation. In order to make the computations more efficient, we have implemented the Algorithm SearchCounterexample using symbolic encodings of graph patterns. We use an appropriate language that is based on first-order predicate logic. As the number of target graph patterns is exponential in the number of nodes in the graph patterns, and symbolic encodings provide a compact representations of large sets and relations, the new encoding can drastically reduce the run-time of the algorithm, especially for complex graph patterns.

The symbolic algorithm can be executed by an interpreter that takes advantage of efficient data structures and algorithms for relational manipulation, such as BDD libraries or SAT solvers. We chose to encode Alg. 1 in the relational programming language RML and execute it using a BDD-based interpreter for that language, namely CrocoPat [4].

Symbolic encoding. The universe of values $\mathcal{U} = \mathcal{N} \cup \mathcal{T}$ is the set of all nodes and types that occur in the system. A variable assignment of a set X of variables is a total function $v : X \to \mathcal{U}$, which assigns to each variable a value from $\mathcal{U}$. The set of all variable assignments of X is denoted by $Val(X)$. For a predicate ϕ over the variables from X, we use $\phi[v]$ to denote the evaluation of ϕ w.r.t. a variable assignment $v \in Val(X)$.

We define the *symbolic representation* of a graph pattern P with k nodes as the tuple $\mathsf{SG}(P) = (X, pn, nn, pe, ne, \phi)$, which consists of the following components: The set $X = \{x_1, \ldots, x_k\}$ is a set of k node variables, each of which represents a node of a concrete graph that matches P. The functions $pn : \mathcal{T} \to 2^X$ and $nn : \mathcal{T} \to 2^X$ assign to each type a set of node variables (empty sets for edge types). A variable $x \in pn(t)$ ($x \in nn(t)$) represents a node of type t that must exist (must not exist) in a matching concrete graph. The functions $pe : \mathcal{T} \to 2^{X \times X}$ and $ne : \mathcal{T} \to 2^{X \times X}$ assign to each type a set of pairs of variables (empty sets for node types). A pair of variables $(x, x') \in pe(t)$ ($(x, x') \in ne(t)$) represents an edge of type t that must exist (must not exist) in a matching concrete graph. The predicate $\phi : Val(X) \to \mathbb{B}$ evaluates to *true* for a given variable assignment v if $\bigwedge_{1 \le i \le k} \bigwedge_{1 \le j \le k \wedge i \neq j} v(x_i) \neq v(x_j)$ (all node variables of the graph pattern are mapped to different nodes) and $\bigwedge_{t \in \mathcal{T}} \bigwedge_{x \in pn(t)} T(x, t)$ (all variables must be of their required node type) and $\phi'[v]$ holds. The predicate ϕ' can be used to include additional properties of the matching graphs into the graph pattern (default: $\phi' := true$). The set of all symbolic repr. of graph patterns is denoted by *GRAPH*.

A graph $G = (N, E)$ *matches* a graph pattern P with k nodes that is given by $\mathsf{SG}(P) = (X, pn, nn, pe, ne, \phi)$, if there exists a variable assignment v of X for which all of the following conditions are fulfilled: (1) the set of nodes $N' = \{v(x_1), \ldots, v(x_k)\}$ is a subset of $\mathcal{N}$, (2) for all nodes $n \in N'$, the node n is element of N and the type of n is t iff $n \in pn(t)$, and the node n is *not* element of N and the type of n is t iff $n \in nn(t)$, (3) for all edges $e \in N' \times N'$, the edge e is element of E and the type of e is t iff $e \in pe(t)$, and the node e is *not* element of E and the type of e is t iff $e \in ne(t)$, and (4) the predicate ϕ is fulfilled: $\phi[v] = true$.

Example 2. **(Graph pattern matching)** Consider the graph pattern P from Fig. 2, which defines the set of all subgraphs of G such that two shuttles are on the same track (the collision accident). The symbolic representation of P is $\mathsf{SG}(P) = (X, pn, nn, pe, ne, \phi)$, where $X = \{x_1, x_2, x_3\}$, $pn = \{(Shuttle, \{x_1, x_2\}), (Track, \{x_3\}), (succ, \emptyset), (on, \emptyset)\}$, $nn = \{(Shuttle, \emptyset), (Track, \emptyset), (succ, \emptyset), (on, \emptyset)\}$, $pe = \{(Shuttle, \emptyset), (Track, \emptyset), (succ, \emptyset), (on, \{(x_1, x_3), (x_2, x_3)\})\}$, $ne = \{(Shuttle, \emptyset), (Track, \emptyset), (succ, \emptyset), (on, \emptyset)\}$, and $\phi = (x_1 \neq x_2) \wedge (x_1 \neq x_3) \wedge (x_2 \neq x_3) \wedge T(x_1, Shuttle) \wedge T(x_2, Shuttle) \wedge T(x_3, Track)$. We now want to check whether the graph G from Example 1 matches pattern P. It matches, because a variable assignment v of X exists: $v(x_1) = sa, v(x_2) = sb, v(x_3) = ta$. The assignment v fulfills all conditions for matching graph G, and the variable assignment specifies a subgraph of G.

In our verification algorithm, we do not consider explicit graphs but graph patterns. Since we can encode graph patterns as first-order predicates over our universe, it is sufficient to consider a universe that contains as many nodes as used in all different graph patterns L_i, R_i, and F_i that are used in the algorithm.

Backward Application. Now we can define the symbolic representation of the backward application of a rule $(L, R)_r$ to a graph pattern $F \cup R$. Let $\mathsf{SG}(L)$, $\mathsf{SG}(R)$, $\mathsf{SG}(F)$, $\mathsf{SG}(L \cap R)$, and $\mathsf{SG}(R \setminus L)$ be the symbolic representations for L, R, F, $L \cap R$, and $R \setminus L$, respectively. As shortcut, we denote the elements of a representation $\mathsf{SG}(P)$ of a graph pattern P by $X_P, pn_P, nn_P, pe_P, ne_P$, and ϕ_P.

We denote the symbolic backward application of $(L,R)_r$ to $F \cup R$ by the function $\mathsf{SBA} : \mathit{GRAPH} \times \mathit{GRAPH} \times \mathit{GRAPH} \rightarrow \mathit{GRAPH}$, with $\mathsf{SBA}(L,R,F) = (X_{LRF}, pn_{LRF}, nn_{LRF}, pe_{LRF}, ne_{LRF}, \phi_{LRF})$. The functions $pn_{LRF}, nn_{LRF}, pe_{LRF}$, and ne_{LRF} are defined as follows: $pn_{LRF}(t) = pn_L(t) \cup pn_F(t)$ (positive nodes from L and F), $nn_{LRF}(t) = nn_L(t) \cup nn_F(t) \cup pn_{R \setminus L}(t)$ (positive nodes from $R \setminus L$), $pe_{LRF}(t) = pe_L(t) \cup pe_F(t)$ (positive edges from L and F), $ne_{LRF}(t) = ne_L(t) \cup ne_F(t) \cup pe_{R \setminus L}(t)$ (positive edges from $R \setminus L$), for all types $t \in \mathcal{T}$, and $X_{LRF} = \bigcup_{t \in \mathcal{T}} pn_{LRF}(t) \cup nn_{LRF}(t)$. The application condition, included into the graph pattern encoding as ϕ', is defined as follows: $\phi'_{LRF} = \bigwedge_{t \in \mathcal{T}} pn_R(t) \cap nn_F(t) = \emptyset \ \wedge \ nn_R(t) \cap pn_F(t) = \emptyset \ \wedge \ \bigwedge_{t \in \mathcal{T}} pe_R(t) \cap ne_F(t) = \emptyset \ \wedge \ ne_R(t) \cap pe_F(t) = \emptyset$, in order to exclude mappings of negative elements on positive elements for R and F, but not for L and F.

Pattern Inclusion. We extend the inclusion operator from patterns to symbolic representations of patterns in the following way: Given two graph patterns P, P' and their encodings $\mathsf{SG}(P') = (X_{P'}, pn_{P'}, nn_{P'}, pe_{P'}, ne_{P'}, \phi_{P'})$ and $\mathsf{SG}(P) = (X_P, pn_P, nn_P, pe_P, ne_P, \phi_P)$, we encode $P' \sqsubseteq P$ by $\mathsf{SG}(P') \sqsubseteq \mathsf{SG}(P)$, i.e., by the function $\sqsubseteq$: $\mathit{GRAPH} \times \mathit{GRAPH} \rightarrow \mathbb{B}$, with $\mathsf{SG}(P') \sqsubseteq \mathsf{SG}(P)$ iff $(\phi_P \Rightarrow \phi'_P) \wedge (\bigwedge_{t \in \mathcal{T}} \mathsf{SUB}(pn_P(t), pn_{P'}(t), nn_P(t), nn_{P'}(t))) \wedge (\bigwedge_{t \in \mathcal{T}} \mathsf{SUB}(pe_P(t), pe_{P'}(t), ne_P(t), ne_{P'}(t)))$, where for sets $X_1, \ldots, X_4$ of variables we define $\mathsf{SUB}(X_1, X_2, X_3, X_4) = (X_2 \subseteq X_1) \wedge (X_4 \subseteq X_3) \wedge (X_1 \cap X_4 = \emptyset) \wedge (X_2 \cap X_3 = \emptyset)$. The conditions ensure that positive and negative elements are not mixed and that all elements of G' are mapped correctly to elements of G.

Symbolic Algorithm. Using the symbolic encoding for the graph pattern that results from the backward application of the rule $(L,R)_r$ to a graph $R \cup F$ and the condition for graph pattern inclusion, $P' \sqsubseteq P$, we can now define the condition to be checked as $\mathsf{SCE}(L,R,F,C) = \mathsf{SG}(C) \sqsubseteq \mathsf{SBA}(\mathsf{SG}(L), \mathsf{SG}(R), \mathsf{SG}(F))$, where $C \in \mathcal{C}$ is a candidate graph pattern.

When checking this condition for different $C_i \in \mathcal{C}$ in parallel, we can further exploit the fact that we can perform these checks using the same universe, as the different C_i do not affect each other during checking. For a rule $(L,R)_r$, a forbidden graph pattern F, and the set of graph patterns $\mathcal{C}$, where X is the set of variables in L , R, and F, and X' is the set of variables in the elements of $\mathcal{C}$, we encode the desired symbolic check as $\forall\ X\ \exists\ X' : \mathsf{SCE}^{\vee}(L,R,F,\mathcal{C})$, where $\mathsf{SCE}^{\vee}(L,R,F,\mathcal{C})$ is the or-combination of the individual checks in the form $\bigvee_{C_i \in \mathcal{C}} \mathsf{SCE}(L,R,F,C_i)$. Any assignment v of the variables X for which $(\nexists\ X' : \mathsf{SCE}^{\vee}(L,R,F,\mathcal{C}))[v]$ is evaluated to *true* represents a counterexample. Algorithm 4 summarizes the resulting symbolic computation steps.

Algorithm 4 SearchCounterexampleSymbolic$(r, F, \mathcal{C})$

```
Input: Rule $(L,R)_r$, GraphPattern $F$, Set⟨GraphPattern⟩ $\mathcal{C}$
Output: Set⟨CounterexampleAssignment⟩
1: return $\{v \in Val(X) \mid \nexists X' : \mathsf{SCE}^{\vee}(L,R,F,\mathcal{C})[v]\}$
```

5. EVALUATION

In order to evaluate the performance and competitiveness of our approach, we have modeled the case study presented in Section 2 with the Fujaba CASE tool and used it to benchmark the following: (1) model checking over graphs, (2) explicit invariant verification using the procedure in Algorithm 3, and (3) symbolic invariant verification using an RML encoding of Algorithm 4.

Case Study Characteristics. Our case study consists of 6 rules, 1 accident, 3 hazards and 15 invariants encoding cardinalities. As we surmise that the performance of our approach is largely dependent on the number and especially the complexity of the individual patterns, we present the main characteristics of the system in Table 1. We list the number of rules (#r) and forbidden graph patterns (#p), and the size of the rules (size(r)) and the forbidden graph patterns (size(p)). The size of a story pattern is given as a tuple $n : e$, where n corresponds to the number of nodes and e corresponds to the number of edges in R, the pattern's right-hand side. All experiments were performed on a Linux machine with a 933 MHz Pentium III processor and 4 GB memory.

Table 1: Characteristics data of the case study

#r	#p	size(r)(n:e)		size(p)(n:e)	
		min	max	min	max
8	19	3:2	7:10	1:1	5:4

Model Checking. We used the GTS model checker GROOVE [21] to carry out the model checking. GROOVE imports GTS specifications and computes all reachable states of the transformation system, optionally bounded by the occurrence of a forbidden graph. Creating an appropriate plug-in, we were able to export our model from Fujaba to GROOVE's input format.

We checked our model, using collision as the forbidden graph, omitting all auxiliary constraints as the assumed topologies imply the required cardinalities. Effectively, models of moderate size can be checked. Applied to small example topologies, the checks provide valuable feedback for the design of a system. Unexpected system behavior can be visualized and analyzed interactively. However, experiments on a topology with 15 tracks confirmed the adverse effect of combinatorial complexity: After moving from 3 (2 min) to 4 (7 min) to 5 (55 min) shuttles, the verification task became intractable. Finally, the obtained result is only valid for the specific topology and initial state used.

Explicit Invariant Checking. We implemented the explicit invariant checking algorithm as a self-contained Fujaba plug-in. It either pronounces a specification correct or produces a set of counterexamples. Checking just the physical part of the case study yielded the expected collisions, whereas the verification of the complete model proved it to be correct.

Figure 10 displays the performance results for the explicit implementation of the SearchCounterexample subroutine. It shows the computation times in seconds for each pair of rule and forbidden graph. The rules are listed on the X-axis, whereas the invariants are listed on the Y-axis. The number of nodes in R has been used to order the rules and invariants. The slope of the graph supports our hypothesis that the computation time of the algorithm mainly depends on the complexity of the involved graphs. The extreme case

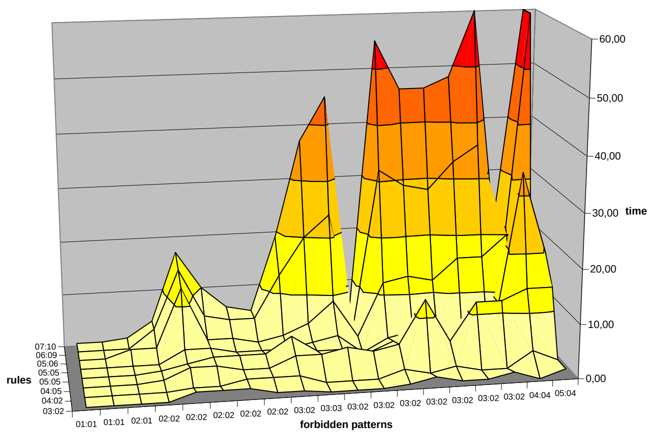


Figure 10: Computation times (in seconds) for explicit search counterexample

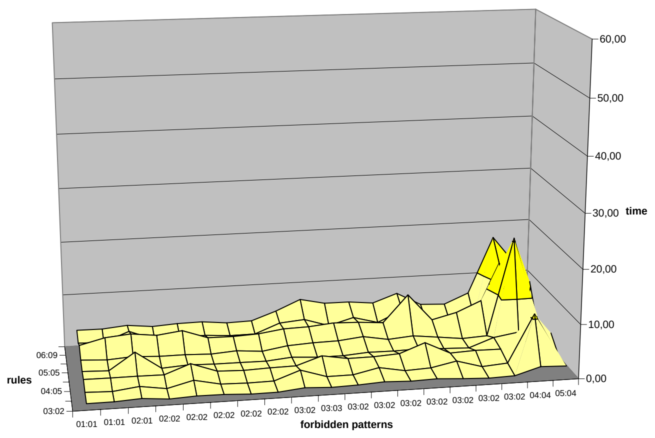


Figure 11: Computation times (in seconds) for symbolic search counterexample

Table 2: Run-times for verifying the largest pairs

Rule / pattern (nodes:edges)	explicit	symbolic
goDC1(7:10) / invalidDCPattern(5:4)	744 s	11.2 s
goDC2(6:09) / invalidDCPattern(5:4)	170 s	6.5 s
goDC1(7:10) / noDC(4:4)	20 s	16.8 s
goDC2(6:09) / noDC(4:4)	7 s	13.1 s
goDC1(7:10) / unambigousOn(3:2)	60 s	6.1 s
goDC2(6:09) / unambigousOn(3:2)	36 s	2.9 s
goDC1(7:10) / unambigousNext(3:2)	48 s	4.1 s
goDC2(6:09) / unambigousNext(3:2)	33 s	2.1 s

resulting from the combination of the goDC1 rule and invalidDCPattern invariant consumed 744s and is omitted from this figure. The explicit algorithm required 34 min for the verification of the overall system.

Symbolic Invariant Checking. The symbolic invariant checker was implemented as a Fujaba plug-in that generates the required RML program from the model and passes it to the graph manipulation engine CrocoPat, which is based on a highly optimized BDD package. The prototypical implementation of the symbolic algorithm results in drastically reduced computation times for large pairs. The run-time increases only moderately with increasing size of the patterns, as shown by Fig. 11. The largest rule/invariant pair is verified in 11.2 s. The symbolic algorithm requires only 5 min to complete the overall verification.

Summary. Table 2 lists the computation times for the largest rule/invariant pairs. The pairs are ordered according to the size of the forbidden graph patterns. For the explicit algorithm, the combinatoric complexity of the rule/invariant pair, i.e., the number of different ways to intersect the patterns, has the most significant impact on the computation time. This explains why the pair goDC2 and noDC is a particularly easy case for the explicit algorithm, in spite of the size of the pair, as the number of possible intersections is constrained by a large number of positive edges.

The symbolic algorithm seems to be dramatically less sensitive to increasing problem complexity of the rules and invariants. The extreme case in the first row in the table, which results from the combination of the goDC1 rule and invalidDCPattern invariant, especially highlights this fact: a run-time of 11.2 s for the symbolic algorithm versus a run-time of 744 s for the explicit algorithm. This case indicates that the speed-up due to the symbolic encoding can be extremely high for certain hard cases with a high number of nodes and edges.

6. RELATED WORK

Verifying finite state spaces. Caporuscio et al. propose to build a system with dynamically changing structures by using architectural patterns [8]. To verify such a system, they suggest to consider a certain minimal sub-model and verify it by model checking. The model checking result has to be generalized manually to ensure that it holds for all possible states. In our approach, in contrast, we support a fully-automatic procedure for verifying whether only safe sub-models can be reached at run-time.

Alloy allows the design and analysis of systems with changing structures [16]. DynAlloy extends Alloy in such a way that state changes can also be modeled [10]. In contrast to our approach, where operations are described by story patterns, Alloy and DynAlloy require operations and properties given as logical formulae. They check whether the given properties are operational invariants of the system.

Varró uses a visual language for modeling systems, and transforms the models then into a model-checker specific input [24]. This approach has been successfully applied to verify service-oriented systems [3]. Instead of transforming a system to a model checker's input format, Rensink performs the model checking directly on the GTS [21]. As a consequence, his tool implementation GROOVE focuses on pure GTS, rather than on object-oriented models. The tool (and language) Real-Time Maude is based on rewriting logics [20]. The tool supports the simulation of a single behavior of the system as well as model checking of the complete state space, if it is finite.

In contrast to our work, the above approaches have two main restrictions. First, they require an initial graph. Second, the application of the methods is only possible if the system to be verified either has a finite state space or the system is abstracted to a finite state model of moderate size. To require a fixed initial graph and a finite state space is not appropriate for mechatronic systems, and the abstraction to a finite state space is (currently) difficult to achieve. However, there is work in progress in this direction (cf. [2]).

Verifying infinite state spaces. The only approach that explicitly addresses the verification of infinite state systems with changing structure so far is the following. A GTS can be transformed into a finite structure, called Petri graph [1]. Such a Petri graph consists of a graph and a Petri net, each of which can be analyzed with existing tools for the analysis of Petri nets. For infinite systems, the authors suggest an approximation. The approach is not appropriate for the verification of mechatronic systems, because it requires an initial graph and the expressiveness of the underlying GTS is rather restricted, e.g., rules must not delete nodes.

Verifying invariants. In the context of graph transformations, Heckel and Wagner used the idea of gluing a rule's RHS with a forbidden graph pattern and then performing a backward application of the rule, to ensure consistency [15]. Using backward application, they transform the forbidden graph patterns into additional negative application conditions (NAC), i.e., negative nodes of the LHS. The transformation of a correct into an incorrect graph is thus prevented by the modified rule. However, the objective of their approach is not to *verify* the consistency of a set of transformation rules w.r.t. a set of forbidden graph patterns, but rather to automatically *construct* a set of modified rules that ensures consistency by avoiding the production of forbidden graphs, at the cost of possibly adding vacuous constraints. Applied to our domain as a run-time checking technique, this approach would only work where the extensions respect the agent's actually accessible context and do not modify physical processes such as the movement of shuttles.

7. CONCLUSION

We presented an extension of the Mechatronic UML development approach towards the modeling and verification of mechatronic multi-agent systems. It takes architecture-level structural adaptation of the agent network at run-time into account. We developed an automated checking technique for structural invariants, based on the formal model of graph transformation systems. System states are modeled as graphs, and system transitions are modeled as graph transformations. Safety properties are modeled as inductive invariants on the graph transformation rules. The performance of our approach is determined by the size of the safety property (given as a set of forbidden graph patterns) and by the size of the transformation rules. The symbolic encoding of the verification procedure enables the application of efficient symbolic graph manipulation engines. Our experiments confirm that the symbolic encoding of our algorithm scales better than the explicit encoding. We recorded a speed-up of more than an order of magnitude for the most complex rule/invariant pair. As expected, both implementations scale vastly better for non-trivial systems than an orthogonal approach that uses state space exploration. Future work in this project includes the improvement of the user-interface to make the interaction between the engineer and the formal model more convenient (e.g., visual counterexample inspection), the evaluation of the approach in an industrial-size case study, and releasing the Fujaba extension as a stable verification product.

8. REFERENCES

[1] P. Baldan, A. Corradini, and B. König. A static analysis technique for graph transformation systems. In *Proc. CONCUR*, LNCS 2154, pages 381–395. Springer, 2001.

[2] P. Baldan, B. König, and A. Rensink. Graph grammar verification through abstraction (summary 2). Proc. Dagstuhl Seminar 04241, 2005.

[3] L. Baresi, R. Heckel, S. Thöne, and D. Varró. Modeling and validation of service-oriented architectures: Application vs. style. In *Proc. ESEC/FSE*, pages 68–77. ACM, 2003.

[4] D. Beyer, A. Noack, and C. Lewerentz. Efficient relational calculation for software analysis. *IEEE Trans. on Software Engineering*, 31(2):137–149, 2005.

[5] R. Bharadwaj and S. Sims. Salsa: Combining constraint solvers with BDDs for automatic invariant checking. In *Proc. TACAS*, LNCS 1785, pages 378–394. Springer, 2000.

[6] D. Bradley, D. Seward, D. Dawson, and S. Burge. *Mechatronics*. Stanley Thornes, 2000.

[7] S. Burmester, H. Giese, and M. Tichy. Model-driven development of reconfigurable mechatronic systems with mechatronic UML. In *Model Driven Architecture: Foundations and Applications*, LNCS 3599, pages 47–61. Springer, 2005.

[8] M. Caporuscio, P. Inverardi, and P. Pelliccione. Formal analysis of architectural patterns. In *Proc. EWSA*, LNCS 3047, pages 10–24. Springer, 2004.

[9] M. Charpentier. Composing invariants. In *Proc. FME*, LNCS 2805, pages 401–421. Springer, 2003.

[10] M. F. Frias, J. P. Galeotti, C. L. Pombo, and N. Aguirre. DynAlloy: Upgrading Alloy with actions. In *Proc. ICSE*, pages 442–451. ACM, 2005.

[11] H. Giese, S. Burmester, W. Schäfer, and O. Oberschelp. Modular design and verification of component-based mechatronic systems with online-reconfiguration. In *Proc. FSE*, pages 179–188. ACM, 2004.

[12] H. Giese and D. Schilling. Towards the automatic verification of inductive invariants for infinite state UML models. Technical Report tr-ri-04-252, University of Paderborn, Germany, 2004.

[13] H. Giese, M. Tichy, S. Burmester, W. Schäfer, and S. Flake. Towards the compositional verification of real-time UML designs. In *Proc. ESEC/FSE*, pages 38–47. ACM, 2003.

[14] R. Heckel, J. Küster, and G. Taentzer. Towards automatic translation of UML models into semantic domains. In *Proc. AGT*, pages 11–22, 2002.

[15] R. Heckel and A. Wagner. Ensuring consistency of conditional graph rewriting — a constructive approach. *ENTCS*, 2, 1995.

[16] D. Jackson. Alloy: A lightweight object modelling notation. *ACM Trans. Software Engineering and Methodology*, 11(2):256–290, 2002.

[17] F. Klein and H. Giese. Separation of concerns for mechatronic multi-agent systems through dynamic communities. In *SELMAS III*, LNCS 3390, pages 272–289. Springer, 2005.

[18] H. Köhler, U. Nickel, J. Niere, and A. Zündorf. Integrating UML diagrams for production control systems. In *Proc. ICSE*, pages 241–251. ACM, 2000.

[19] D. Musliner, R. Goldman, M. Pelican, and K. Krebsbach. Self-adaptive software for hard real-time environments. *IEEE Intelligent Systems*, 14(4), 1999.

[20] P. Ölveczky and J. Meseguer. Specification and analysis of real-time systems using Real-Time Maude. In *Proc. FASE*, LNCS 2984, pages 354–358. Springer, 2004.

[21] A. Rensink. Towards model checking graph grammars. In *Proc. AVoCS*, pages 150–160. University of Southampton, 2003.

[22] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation: Foundations*, volume 1. World Scientific Pub Co, 1997.

[23] J. Sztipanovits, G. Karsai, and T. Bapty. Self-adaptive software for signal processing. *Commun. ACM*, 41(5):66–73, 1998.

[24] D. Varró. Automated formal verification of visual modeling languages by model checking. *Software and System Modeling*, 3(2):85–113, 2004.