Symbolic Invariant Verification for Systems with Dynamic Structural Adaptation

Holger Giese     Daniela Schilling
Florian Klein     Basil Becker
Software Engineering Group
University of Paderborn
D-33098 Paderborn, Germany

Dirk Beyer
Models and Theory of Computation Group
Ecole Polytechnique Fédérale de Lausanne
CH-1015 Lausanne, Switzerland

ABSTRACT
The next generation of networked mechatronic systems will be characterized by complex coordination and structural adaptation at runtime. Crucial safety properties have to be guaranteed for all potential structural configurations. Testing cannot provide safety guarantees, while model checking and theorem proving do not scale for such systems. We present an approach for the verification of arbitrarily large multi-agent systems featuring complex coordination and structural adaptation from the mechatronic domain. We overcome the limitations of existing techniques by exploiting the local character of the 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. 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.2.2 [Software Engineering]: Design Tools and Techniques—Computer-aided software engineering (CASE); D.2.4 [Software Engineering]: Software/Program Verification; D.2.11 [Software Engineering]: Software Architectures—domain-specific architectures, patterns

General Terms
Design, Reliability, Verification

Keywords
Formal verification, Graph transformation systems, Structural invariants, Embedded systems, Mechatronics

1. INTRODUCTION
Mechatronic systems combine traditional mechanical and electrical engineering with technologies from software engineering in order provide dependable technical solutions to complex real-world problems. In the future, advanced mechatronic multi-agent systems are expected to exhibit increasingly complex context-dependent behavior. The key to enhancing 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 envisaged systems will not only exhibit self-adaptive behavior (cf. [23, 19]) 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 which is reachable at runtime 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 runtime. Techniques such as testing the system in several test environments and its operation environment alone are not sufficient for this task due to their incomplete nature. On the other hand, complete automated verification techniques do not scale: model checking can prove safety properties for models of moderate size only, and semi-automatic approaches such as theorem proving require advanced proof skills that are usually not available.

Previous approaches have already used model-checking techniques for the verification of systems with structural adaptation. However, they are all limited to finite state models of rather small size [14, 24, 3, 21, 20, 16]. What is even more problematic is that they require that a small initial system topology is known, 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, a case which is not covered either so far.

We present a verification technique that can handle the real-time coordination and structural adaptation of arbitrarily large multi-agent systems in 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 (cf. [15])—. Safety requirements of the system are also 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 any 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 do we want to 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 were not yet 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 verification of 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[1] in the modeling and in the evaluation section.

To provide a tool implementation that scales to large systems, we developed—besides the traditional explicit algorithm that 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 implemented both the explicit and the symbolic algorithm and integrated them into the UML CASE tool Fujaba[2]. We adapted the GROOVE engine [21] for comparing the results with 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 Section 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 Section 4, where we also discuss explicit and symbolic algorithms for the verification. We have implemented both algorithms in our tool framework and present the results of our performance experiments in Section 5. Section 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 runtime. 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 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 struc-

http://www.railcab.de
http://www.fujaba.de
2.2 Modeling

All our notations are based on the UML. Besides the standard notations, we use MECHATRONIC UML [7, 13, 11] for the specification of real-time coordination patterns. We also employ story patterns, an extended type of UML collaboration diagrams based on the theory of graph transformations 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.

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 Figure 2).

Figure 2: Diagram specifying a Collision accident

Note that cardinalities in conditions are not currently 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 certain stereotypes, the pre- and postcondition can be compactly specified as a single object graph where unmarked elements remain constant, red elements annotated with «destroy» are erased and green 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.

We use story patterns for expressing all kinds of state changes, e.g., agent behavior, rules, and physical processes. The story patterns in Figure 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.

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 MECHA-
TRONIC 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 Shuttle Culture 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 DistanceCoordinationPattern as a subculture. The DistanceCoordinationPattern, which instantiates that culture, groups two Shuttles which take on the rear respectively front role, again marked by stereotypes.

The Shuttle Culture introduces two behavioral rules: goSimple1 (see Figure 8) 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.

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 Figure 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.

Figure 8: Behavioral rule: Coordinated movement

The behavioral rules goDC1 (see Figure 9) 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 9: Invariant: No uncoordinated movement of Shuttles in close proximity

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. Though not required for the operational correctness of the model, this implied condition (see Figure 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.

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 underlying formal model of graph transformation systems in general, and of story patterns in particular, as prerequisite for the verification of safety properties.

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

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 of nodes \( N = \{sa, sb, ta\}\), the set of edges \( E = N \times N\), the set of types \( T = \{Shuttle, Track, on\}\), and the type relation \( T \) with \((sa, Shuttle) \in T, (sb, Shuttle) \in T, (ta, Track) \in T, ((ta, sa), on) \in T, ((ta, sb), on) \in T\). Then \( G = (N, \{(ta, sa), (ta, sb)\})\) is the graph representing the collision accident depicted in Figure 2.

**Graph Patterns.** A graph pattern \( P = (N^+, N^-, E^+, E^-)\) consists of sets of positive and negative nodes \( N^+ \) and \( N^-\), and sets of positive and negative edges \( E^+ \) and \( E^-\). The sets \( E^+ \) and \( E^-\) are subsets of \((N^+ \cup N^-) \times (N^+ \cup N^-)\). We further denote \( P \) restricted to \( N^+ \) and \( E^+\) as \( P^+\). A graph pattern defines a 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 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 with the same type, respectively. There can be an arbitrary number of negative elements, but negative node 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 \( P \subseteq P'\).

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 evolution, whereas a forbidden pattern must never be fulfilled (hazard, accident).

**Graph Transformation Systems.** A graph transformation transforms a graph into another 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 evolution of the system.

**Story patterns** can be seen as extended graph patterns that allow annotating graph elements with \( \ll create \rr \) and \( \ll destroy \rr \). As introduced informally above, they may be used for specifying transformation rules. A graph transformation rule \(( L, R),\) 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 \( \ll create \rr \), including negative elements, whereas \( R \) consists of all elements not annotated with \( \ll destroy \rr \). The elements annotated with \( \ll create \rr \) will be created by the rule (from \( R^+ \setminus L^+\)), while those annotated with \( \ll destroy \rr \) will be deleted (from \( L^+ \setminus R^+\)). Unannotated elements are preserved by the application (from \( L^+ \cap R^+\)).

A graph transformation system (GTS) \( S = (R, prio)\) consists of a set of graph transformation rules \( R \) (defined by story patterns), defining all possible transformations in the transformation system, and a priority function \( prio : R \rightarrow N\), which assigns a priority to each rule (higher numbers take precedence). An additional set of initial graphs may describe all possible initial states of the system.

A rule \(( L, R),\) is applicable to a graph \( G \) if \( G \) matches \( L \) and \( G \) does not match the LHS \( L'\) of any other rule \(( L', R'),\) with higher priority. During the application of a rule \(( L, R),\) 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 \( G\). We write \( G \gg \approx \rightarrow G'\) if rule \( r\) can be applied to graph \( G\) and the application results in graph \( G'\). We write \( G \gg \approx \rightarrow G'\) if \( G\) is transformed into \( G'\) by a sequence of rule applications. Given a graph transformation system \( S\) and a graph \( G\), the set of states reachable by applying rules from \( S\) to \( G\) is denoted as \( REACH(S, G) = \{G' | G \gg \approx \rightarrow G'\}\).

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 \(( L, R),\) to a graph pattern \( P \) is defined as \( \text{ApplyBack}(r, P) := P \setminus (R \setminus L) \cup L\) if \( R \subseteq P \land (L \setminus R) \cap P = G_0\) holds, where \( G_0 = (\emptyset, \emptyset)\) is the empty graph.

4Our notion of application is a restricted version of the Single Pushout approach (cf. Figure 22). The approach is restricted such 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. \( R\).
scribed 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 gain efficiency by running it on an interpreter that is optimized for such algorithms.

**Verification Problem.** Safety-violation conditions (hazards, accidents) are represented by a set of forbidden graph patterns $F = \{F_1, \ldots, F_n\}$. The graph $G$ fulfills the safety property $\Phi$, denoted as $G \models \Phi$, if $G$ matches none of the graph patterns in $F$. If an $F \in F$ that is matched by $G$ exists, we call it a witness for the property $\neg \Phi$. The property $\Phi_F$ is an operational invariant of a GTS $S$ if and only if $G \models \Phi_F$ for all $G \in \text{REACH}(S, G^0)$, for a given initial graph $G^0$ (cf. [9]).

Operational invariants in general cannot be fully automatically checked, as graph transformation systems with types are Turing complete. Therefore it is usually not possible to use automatic verification techniques (cf. [2]). Only for the restricted case where $G^0$ and $\text{REACH}(S, G^0)$ are finite, available model checking approaches for GTS can be employed.

The property $\Phi_F$ is an inductive invariant of a GTS $S = (R, \text{pre})$ if and only if for all graphs $G$ and for all rules $r \in R$ the following holds: $G \models \Phi_F$ and $G \rightarrow_r G'$ implies $G' \models \Phi_F$. If this is the case and the initial graph $G^0$ fulfills the property, then $\Phi_F$ is also an operational invariant (the vice versa is not true in general).

As we cannot place undue restrictions on the system’s deployment at design time, the potential initial state $G^0$ is usually not known. In addition, the reachable state space $\text{REACH}(S, G^0)$ may not always be finite. Therefore, we propose to look for inductive invariants of the GTS and present in the next subsection our verification approach for them which covers infinite state systems and arbitrary but correct initial deployments (cf. [12]).

**Checking Inductive Invariants.** We solve the verification problem by checking whether the property $\Phi_F$ is an inductive invariant of the GTS. Reformulating the definition of an inductive invariant: a property $\Phi_F$ is an inductive invariant of a GTS $S$ if and only if for all graphs $G$ and for all rules $r \in R$ such that $G \models \Phi_F$ and $G \rightarrow_r G'$ implies $G' \models \Phi_F$. We further call such a pair $(G, r)$ a counterexample, which witnesses the violation of property $\Phi_F$ 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 the small part of a graph that is matched by the rule can be affected by the rule application. For a counterexample $(G, r)$, the local modification of $G$ by rule $r$ must be responsible for transforming the correct graph $G$ into a graph that violates the property. A first case where a forbidden graph can result from a rule application occurs when the rule completes the positive part of a forbidden graph. The second case occurs if elements of $G$ are deleted that ensured that the graph $G$ was not matched by any forbidden graph pattern.

Any possible counterexample must fall into either case 1, 2, or both, and thus we can conclude that $G'$ must contain an intersection between the positive or negative nodes of the RHS of a rule and a forbidden graph. Consequently, we can check that no counterexample exists (and thus that $\Phi_F$ is an inductive invariant) by considering the finite set $\Theta$ of graph patterns $P'$ that are combinations of a RHS $R$ of a rule $r$ and a forbidden graph pattern $F \in F$, formally:

$$\Theta(F, R) = \{\text{iso}(F) \cup R | \exists \text{iso}(F) : \text{iso}(F) \cap R \neq \emptyset\}.$$ For each of these combined graph patterns $P'$ we only have to check whether $G \models \Phi_F$ with $P \subseteq P'$ for the graph pattern $P$ with $P \Rightarrow P'$. Let $S = (R, \text{pre})$ be a GTS, $P'$ be a graph pattern, and $r$ be a transformation rule of $S$ ($r \in R$). The pair $(P, r)$ is a counterexample for $\Phi_F$ if the following conditions hold:

1. $P' \in \Theta(F, R)$ for some $F \in F$, and
2. $P \Rightarrow P'$, i.e., the rule $r$ can be applied to graph pattern $P$ and the resulting graph pattern is $P'$ (this implies that no $r' \in R \setminus \{r\}$ exists with $\text{prio}(r') > \text{prio}(r)$ that matches $G$ due to the definition of rule application), and
3. $P' \not\subseteq F$ for all graph patterns $P' \not\subseteq P$. i.e., no witness for $\neg \Phi_F$ could be found.

We can then use the above conditions to check whether a counterexample exists. Algorithm 1 checks for a given rule $(L, R) \in \mathcal{R}$ and a forbidden graph pattern $F \in F$ whether a counterexample exists. The algorithm first computes the set of all possible target graph patterns $(\text{tgSet})$ for $R$ and the forbidden graph pattern $F$ (using function $\Theta$). Then the related source graph patterns $(\text{sgpSet})$ are determined using $\text{ApplyBack}$. The set of candidates of graph-patterns $C$ is given as the set of forbidden graph patterns $F$ plus the LHS graph patterns of all rules with higher priority. The occurrence of a graph pattern from $C$ in the source graph pattern then means that the transformation is safe, i.e., the source graph pattern is already forbidden or the application of $r$ was not possible due to another rule with higher priority. One considered source graph pattern together with the rule $r$ is included in the resulting set of counterexamples, if the source graph pattern does not match any graph pattern from $C$.

Algorithm 2 checks that $\Phi_F$ is an inductive invariant by enumerating the combinations of rules $r \in R$ and forbidden graph patterns $F \in F$. For each combination, it is checked whether a counterexample exists using an Algorithm SearchCounterexample. The Algorithm SearchCounterexample will return the empty set if no counterexample exists, otherwise it returns a non-empty set of counterexamples (the implementation in Algorithm 1 returns all counterexamples). The property $\Phi_F$ is an inductive invariant, if the algorithm does not find any counterexample.

5In the case of negative elements in the rules or graph patterns, the check is a sufficient but not a necessary criteria for a counterexample while without negative elements the check is sufficient and necessary. This limitation holds due to rare cases, where the identified source graph pattern may not include a forbidden graph pattern due missing negative elements but no concrete counterexample graph graph which matches the source graph pattern can be constructed which does not match any forbidden graph pattern.
**Algorithm 2** CheckInvariant(S, F)

Input: GTS $S = (R, p)$, Set(GraphPattern) $F$
Output: Set(Counterexample)

Variables: Rule $r$, Set(GraphPattern) candSet, GraphPattern $F$

1: for all $(L, r) \in R$ do  
2: $candSet := F \cup \{L|(U, R), v \in R \land prio(r') > prio(r)\}$  
3: for all $F \in F$ do  
4: $res := SearchCounterexample(r, F, candSet)$  
5: if $res \neq \emptyset$ then  
6: // A counterexample pattern has been found  
7: return $res$  
8: end if  
9: end for  
10: end for  
11: return $\emptyset$

**Explicit implementation.** Algorithm 2 shows an implementation for explicitly represented graphs (explicit enumeration). A serious problem of the explicit algorithm is that the for all loop in line 1 of the Algorithm SearchCounterexampleExplicit is highly exponential w.r.t. the size of the graphs involved since all target graph patterns (elements from $\Theta(F, R)$) have to be determined and enumerated.

**Algorithm 3** SearchCounterexampleExplicit(r, $P, C$)

Input: Rule $(L, R)$, GraphPattern $F$, Set(GraphPattern) $C$
Output: Set(Counterexample)

Variables: GraphPattern $target$, source, cand, GraphPattern $F$

1: for all target $\in \Theta(F, R)$ do  
2: $source := ApplyBack(r, target)$  
3: $correct := true$  
4: $candSet := \emptyset$  
5: while $correct \land candSet \neq \emptyset$ do  
6: choose cand $\in candSet$  
7: $candSet := candSet \setminus \{cand\}$  
8: if cand $\subseteq$ source then  
9: $correct := false$  
10: end if  
11: end while  
12: if correct then  
13: // A counterexample pattern has been found  
14: return $\{(source, r)\}$  
15: end if  
16: end for  
17: return $\emptyset$

**Symbolic implementation.** For more efficient computation, we use a symbolic encoding of Algorithm SearchCounterexampleExplicit in a language that is based on first-order predicate calculus. Thus, we can avoid an algorithm that explicitly enumerates the possibly exponentially many target graph patterns a priori. Such a symbolic algorithm can then 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 use a symbolic encoding of Algorithm 3 in the relational programming language RML, and execute our programs with a BDD-based interpreter for that language [4].

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

We define the symbolic representation of a graph pattern $P$ with $k$ nodes as the tuple $SG(P) = (X, pn, nn, pe, ne, \phi)$, 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 $pn : T \rightarrow 2^X$ and $nn : T \rightarrow 2^X$ 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 $pe : T \rightarrow 2^{X \times X}$ and $ne : T \rightarrow 2^{X \times X}$ 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 set of all variable representations of graph patterns is denoted as $GRAPH$. A graph $G = (N, E)$ matches a graph pattern $P$ with $k$ nodes that is given by $SG(P) = (X, pn, nn, pe, ne, \phi)$, if there exists a variable assignment $\nu$ of $X$ that fulfills all of the following conditions: (1) the set of nodes $N' = \{v(x_1), \ldots, v(x_k)\}$ is a subset of $N$, (2) for all nodes $n \in N'$, the node $n$ is element of $N$ and the type of $n$ is $t$ if $n \in pn(t)$, and the node $n$ is not element of $N$ and the type of $n$ is $t$ if $n \in nn(t)$, (3) for all edges $e \in N' \times N'$, the edge $e$ is element of $E$ and the type of $e$ is $t$ if $e \in pe(t)$, and the edge $e$ is not element of $E$ and the type of $e$ is $t$ if $e \in ne(t)$, and (4) the variable assignment $\nu$ fulfills the formula $\bigwedge_{1 \leq i < j \leq k} \bigwedge_{\phi \in GraphPattern} \nu(x_i) \neq \nu(x_j)$ (all node variables of a considered graph pattern are mapped to different nodes) which together with $\bigwedge_{e \in E} \bigwedge_{\phi \in GraphPattern} T(x, t)$ (all variables must be of their respective node type) forms the predicate $\phi$.

**Example 2. (Graph pattern matching)** Consider the graph pattern $P$ from Figure 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 $SG(P) = (X, pn, nn, pe, ne, \phi)$, where $X = \{x_1, x_2, x_3\}$, $pn = \{(Shuttle, [x_1, x_2]), (Track, [x_3]), (succ, \emptyset), (on, \emptyset)\}$, $nn = \{(Shuttle, \emptyset), (Track, \emptyset), (succ, \emptyset), (on, \emptyset)\}$, $pe = \{(Shuttle, \emptyset), (Track, \emptyset), (succ, \emptyset), (on, \{x_1, x_3\})\}$, and $\phi = T(x_1, Shuttle) \land (T(x_2, Shuttle) \lor (T(x_3, Track) \land x_1 \neq x_2)$. Now we want to check whether the graph $G$ from the previous Example matches graph pattern $P$. It matches, because there exists a variable assignment $\nu$ of $X$ that fulfills all conditions for matching graph $G$, and the variable assignment specifies a subgraph of $G$, accordingly: $\nu(x_1) = sa, \nu(x_2) = sb, \nu(x_3) = ta$.

In our verification algorithm, we do not want to 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 the different graph patterns $L_1$, $R_1$, and $F_1$, where $L_1$, $R_1$, and $F_1$ are the graph patterns that are used in the algorithm.

**Backward Application.** Now we can define the symbolic representation of the backward application of a rule $r \in R$. To a graph pattern $F \cup R$. Let $SG(L)$, $SG(R)$, $SG(F)$, $SG(L \cup R)$, and $SG(R \setminus L)$ be the symbolic representations for $L$, $R$, $F$, $L \cup R$, and $R \setminus L$, respectively. As shortcut, we denote the
elements of a representation $SG(P)$ of a graph pattern $P$ as $X_P, pn_P, nn_P, pe_P, ne_P,$ and $\phi_P$. The application condition $\phi_{L,R,F}$ is defined as

$$\bigwedge_{\nu \in \mathcal{G}} pn_{\nu}(t) \land nn_{\nu}(t) = \emptyset \land \phi_P \land \bigwedge_{\nu \in \mathcal{G}} pe_{\nu}(t) \land ne_{\nu}(t) = \emptyset \land \phi_P$$

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 combination of the variables of $\mathcal{G}$.

Symbolic Algorithm. Using the symbolic encoding for the graph pattern which results from the backward application of the rule $(L, R)$ to a graph $R \cup F$ and the condition of pattern inclusion $P' \subseteq P$, we can derive the single condition representing the fact $P' \subseteq P$ using a function $\text{SINCL} : \mathcal{G} \times \mathcal{G} \rightarrow \text{BOOL}$ as follows: $\text{SINCL}(\mathcal{G}(P'), \mathcal{G}(P)) := \phi_P \land \phi_{L,R,F} \land \left( \bigwedge_{\nu \in \mathcal{G}} \text{SUB}(pn_{\nu}(t), pn_{\nu}(t), pn_{\nu}(t), pn_{\nu}(t)) \right) \land \left( \bigwedge_{\nu \in \mathcal{G}} \text{SUB}(pe_{\nu}(t), pe_{\nu}(t), pe_{\nu}(t), \phi_P) \right)$, where for any sets $X_1, \ldots, X_4$ of variable sets we define $\text{SUB}(X_1, X_2, X_3, X_4) := X_2 \subseteq X_1 \land X_4 \subseteq X_3 \land (X_1 \cup X_4 = \emptyset) \land (X_2 \cap X_3 = \emptyset)$. The conditions ensure that positive and negative elements not mixed and that all elements of $G'$ are mapped correctly to elements of $G$.

Pattern Inclusion. Given encodings for the graph patterns $P$ and $P'$ in form of $\mathcal{G}(P) = (\psi_p, \psi_{pn}, \psi_{nn}, \psi_{pe}, \psi_{ne}, \phi_P)$ and $\mathcal{G}(P) = (\psi_{P'}, \psi_{pn}, \psi_{nn}, \psi_{pe}, \psi_{ne}, \phi_P)$, we can derive a single condition representing the fact $P' \subseteq P$ using a function $\text{SINCL} : \mathcal{G} \times \mathcal{G} \rightarrow \text{BOOL}$ as follows: $\text{SINCL}(\mathcal{G}(P'), \mathcal{G}(P)) := \phi_P \land \phi_{P', R, F} \land \left( \bigwedge_{\nu \in \mathcal{G}} \text{SUB}(pn_{\nu}(t), pn_{\nu}(t), pn_{\nu}(t), pn_{\nu}(t)) \right) \land \left( \bigwedge_{\nu \in \mathcal{G}} \text{SUB}(pe_{\nu}(t), pe_{\nu}(t), pe_{\nu}(t), \phi_P) \right)$, where for any sets $X_1, \ldots, X_4$ of variable sets we define $\text{SUB}(X_1, X_2, X_3, X_4) := X_2 \subseteq X_1 \land X_4 \subseteq X_3 \land (X_1 \cup X_4 = \emptyset) \land (X_2 \cap X_3 = \emptyset)$. The conditions ensure that positive and negative elements not mixed and that all elements of $G'$ are mapped correctly to elements of $G$.

5. EVALUATION

In order to evaluate the performance and competitiveness of our approach, we have modelled the case study presented in Section 2 with the Fajuba 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 two 933 GHz Pentium III processors and 4 GB memory.

Table 1: Characteristics data of the case study

<table>
<thead>
<tr>
<th>#r</th>
<th>#p</th>
<th>size($r$)(node)</th>
<th>size($p$)(node)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>19</td>
<td>3.2</td>
<td>5.6</td>
</tr>
<tr>
<td>7</td>
<td>10</td>
<td>5.1</td>
<td>7.10</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>1.1</td>
<td>5.4</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>3.2</td>
<td>3.2</td>
</tr>
</tbody>
</table>

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 Fajuba to GROOVE’s input format.

We checked our model, using collision as the forbidden graph. Effectively, models of moderate size can be checked. Applied to small example topologies, the checks provide valuable feedback for the design of a system. 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 Fajuba plugin. 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. The explicit algorithm required 34.0 min for the verification.

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 invariant. The slope of the graph clearly supports our hypothesis that the computation time for the routines mainly depends on the complexity of the involved graphs.
6. RELATED WORK

Verifying finite state spaces. In the literature, there are some approaches that can be used to verify systems with changing structure but finite state space. Similarly to the presented approach, Caporuscio et. al. \[8\] propose to build a system with dynamic changing structures by using architectural patterns. To verify such a system, they suggest to consider a certain minimal sub model and model check it. The model checking result then has to be manually generalized to ensure that it holds for all possible states. In the presented approach, in contrast, we support automatically checking whether only verified submodels can be reached at runtime.

Alloy \[10\] allows designing and analyzing systems with changing structures. DynAlloy \[10\] extends Alloy in such a way that state changes can also be modeled. 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.

In \[24\], systems specified by a visual language are transformed to model checker specific inputs. The model checker then verifies the system and, in case of an erroneous system, delivers a counter-example. This approach has been successfully used to verify service-oriented systems in \[8\].

Instead of transforming a system to a model checker's input, Rensink \[21\] suggests performing model checking directly on GTS. As a consequence, the corresponding GROOVE tool focuses on pure GTS, rather object-orientation.

The language of Real-Time Maude \[20\] is based on rewriting logics. The tool supports the simulation of a single behavior of the system as well as model checking the complete state space if it is finite.

In contrast to our approach, the approaches in \[16\] \[10\] \[24\] \[3\] \[21\] \[20\] have two main restrictions. At first they require an initial graph. Secondly, a verification is only possible if the system to be verified has a finite state space of moderate size. Otherwise the state space has to be cut down to a finite one which results in imprecise results. But these two restrictions are not appropriate when analyzing mechatronic systems.

Verifying infinite state spaces. Approaches which have been developed to verify infinite state systems with changing structure are rather rare:

In \[1\] and former publications, an approach is presented where a possibly infinite GTS is transformed into a finite structure, called Petri graph. Such a Petri graph consists of a graph and a Petri net, each of which can be analyzed with existing tools. For infinite systems, the authors suggest an approximation. The approach is not appropriate for the verification of mechatronic systems as an initial graph is required and the expressibility of the underlying GTS is rather restricted, e.g. rules must not delete nodes.

The idea of gluing a rule’s RHS with a forbidden graph pattern and then performing a backwards application of the rule was formerly presented by Heckel and Wagner in \[15\]. By applying the rules backwards, the forbidden graph patterns are transformed into additional graphs of the rule’s NAC. Applying the modified rule to a correct graph results in a correct graph again. However, the modification is performed separately for each consistency constraint. If
a consistency constraint can only be invalidated by a rule if it is applied to a graph invalidating another constraint, this can lead to unnecessary changes of the rules. I.e., the rule is modified although our approach returned that it was correct before. Another disadvantage is that each of these unnecessary extensions of a rule’s NAC has to be checked at run time.

7. CONCLUSION

We presented an extension of the Mechatronic UML development approach towards the modeling and verification of mechatronic multi-agent systems. It takes architectural-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 evolution is modeled as graph transformation. Safety properties are modeled as inductive invariants on the graph transformation rules. Therefore, the performance of our approach is determined by the size of the safety property (given as set of 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 show that the symbolic encoding of the problem scales much better than the explicit algorithm, and also as an orthogonal approach that uses state space exploration. In our experiments, the verification of the largest rule against the largest forbidden graph pattern could be solved in 12s, while the explicit algorithm needed more than 12 min for the same task. The 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


