Temporal Reconfiguration Plans for Self-Adaptive Systems*

Steffen Ziegert and Heike Wehrheim

Department of Computer Science
University of Paderborn
33098 Paderborn, Germany
{steffen.ziegert, wehrheim}@uni-paderborn.de

Abstract: Self-adaptive systems have the ability to autonomously reconfigure their structure in order to meet specific goals. Such systems often include a *planning* component, computing plans of reconfiguration steps. However, despite the fact that reconfigurations take time in reality, most planning approaches for self-adaptive systems are non-temporal.

In this paper, we present a model-based approach to the generation of *temporal* reconfiguration plans. Besides allowing for durative reconfigurations, our technique also neatly solves concurrency issues arising in such a temporal setting. This provides the modeller with a clear and easy-to-use basis for modelling while at the same time giving an automatic method for plan construction.

1 Introduction

The development of self-adaptive systems belongs to the most complex tasks in software engineering today and requires a rigorous model-driven approach. Self-adaptive systems possess the ability to accommodate to changes in their environment at runtime. Such changes may include the failure of system components, the activities of other agents in the environment, or changing user demands.

Self-adapting to a resulting new situation, calls for a number of reconfigurations that have to be executed. In general, such reconfigurations do not have to be restricted to components' internal states, but may include changes of the system's architecture like the creation and deletion of software component instances or communication links between them.

Systems that decide when and how to adapt their architecture, are said to have a self-organizing [GMK02] or self-managing [BCDW04] architecture. Kramer and Magee [KM07] defined a reference architectural model with three layers for the development of such systems. It consists of a goal management, a change management, and a component control layer. The goal management layer accomplishes time-consuming tasks, like the computation of reconfiguration plans for given goal specifications. The resulting reconfiguration plans state how to adapt to the environment, i.e. which reconfigurations to apply,

*This work was developed in the course of the Collaborative Research Centre 614 "Self-optimizing Concepts and Structures in Mechanical Engineering" and funded by the German Research Foundation (DFG).

and when to execute them. The second layer, the change management layer, provides the capabilities to adapt the system's architecture. The component control layer is the bottom layer and accomplishes the basic tasks of the system by providing the implementation of primitive features. In this paper, we are specifically concerned with the top layer, i.e. with the creation of *reconfiguration plans*.

We employ a model-based approach to the design of self-adaptive systems. Reconfigurations concern the architecture of the system. System models are given in MECHATRONICUML [BBB+12], a UML profile for the development of self-adaptive mechatronic systems. Here, (initial as well as other) system architectures are modelled as *graphs*, and reconfigurations as story patterns [FNTZ98], a formalism based on *graph transformations*, which schematically define how an architecture configuration can be transformed into a new one. Graph transformations are frequently used for specifying the reconfigurations of self-adaptive systems, e.g. [LM98, WF02], and enable verification techniques to be applied [BBG+06, SWW11]. However, only few apply planning techniques on these models, e.g. [EW11], to actually generate reconfiguration plans that achieve the system's goals. However, this is needed for the entire reconfiguration process to happen autonomously, cf. [BCDW04].

The reconfigurations of a system architecture consume *time* and might be carried out *in parallel*. In fact, in some application domains it might be counterintuitive to execute the reconfigurations sequentially, because they refer to highly independent components (or even different agents in multi-agent systems). However, most current planning approaches to architectural reconfiguration, do not support time, and consequently only generate non-temporal plans. The only one that does, is [TK11], an approach that also builds on graph transformation rules for modelling reconfigurations. However, their approach to assign annotations like «at_start» and «at_end» as stereotypes to the elements of rules is cumbersome from a modelling perspective: possible interferences force the rules to be precisely formulated.

The approach we follow here is an extension of [TK11]. We solve planning tasks by translating (parts of) our models into the Planning Domain Definition Language (PDDL) and feeding them into an off-the-shelf planning system, like SGPlan_6 [CWH06]. However, our solution renounces from assigning start and end annotations. Instead, we apply a *locking mechanism* that is based on the idea of restricting read or write access to elements, when they are in use by a reconfiguration. This is similar to the concurrency control methods implemented by database management systems. Unintended interferences, e.g. the deinstantiation and use of a software component at the same time, are thus avoided, without the need for a modeller to specify this. Apart from a model-based approach for planning time-consuming architectural reconfigurations, our approach can also be seen as a knowledge engineering contribution to the PDDL community.

2 Application Scenario

Our running example is based on the RailCab project that is developed at the University of Paderborn. The RailCab project consists of autonomous, driverless shuttles, called

RailCabs, that operate on a railway system. Each RailCab has an individual goal, e.g. transporting passengers or goods to a specified target station. An important feature of the project is the RailCabs' ability to drive in convoy mode, i.e. RailCabs can minimize the air gap between each other to safe energy. To safely operate in a convoy, acceleration and braking has to be coordinated and managed between convoy members. The instantiation and deinstantiation of software components for the communication and coordination of RailCabs is one example for the high dynamics of the system's communication structure. For the purpose of this paper, we keep the application example small and do not cover every aspect of the system's communication structure.

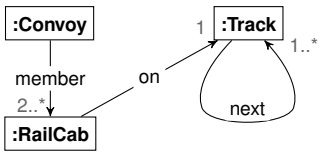

Figure 1: Class diagram of our (simplified) RailCab scenario

We begin with the specification of the system in MECHATRONICUML by first modelling its structure as a class diagram (see Figure 1). The railway system consists of track segments that are connected to each other via `next` links. A RailCab that operates in the system can occupy such a track segment. Furthermore, RailCabs can coordinate with other RailCabs to form a convoy. Such an active convoy operation is represented by an instance of the `Convoy` type. A `Convoy` instance has a `member` link to each participating RailCab.

In MECHATRONICUML, the behaviour of components – more precisely their roles – is modelled with real-time state charts, which allow the definition of communication and timing constraints. Structural reconfigurations, e.g. the instantiation of a convoy, are modelled with story patterns, an extension of UML object diagrams. Since our contribution deals with modelling the structural reconfiguration of self-adaptive systems, the internal behaviour of components is not considered here.

To model reconfiguration actions of our example system, we use story patterns that are typed over the class diagram shown in Figure 1. Story patterns have a formal semantics[1] based on (typed) graph transformation systems [EHK+97]. A graph transformation system consists of a graph representing the initial configuration of the system and a set of rules. Each rule consists of a pair of graphs, called left-hand side (LHS) and right-hand side (RHS), that schematically define how the graph representing the system's configuration can be transformed into new configurations. Elements that are specified in both graphs are preserved, other elements are deleted (if specified in the LHS only) or created (if specified in the RHS only). Syntactically, a story pattern represents such a rule by integrating the LHS and RHS into one graph and using stereotypes to indicate elements that are only present in the LHS or in the RHS [FNTZ98]. The graph-like representation allows not only for an intuitive modelling but also harmonizes with modern approaches for the development of self-optimizing mechatronical systems, e.g. [THHO08].

Figure 2 provides an example of a reconfiguration which takes 4 time units: a RailCab

[1] Story patterns follow the single pushout approach to graph transformation.

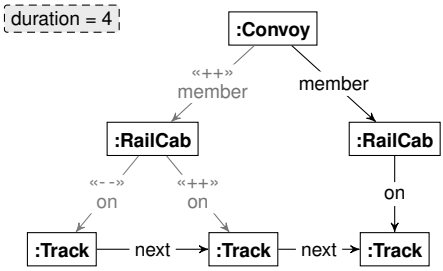

Figure 2: Story pattern `joinConvoy`

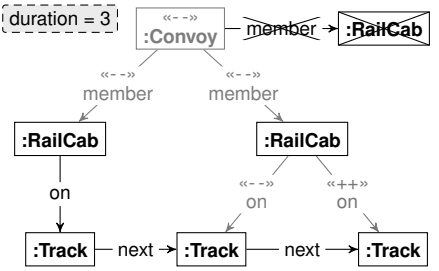

Figure 3: Story pattern `breakConvoy`

joining a convoy of RailCabs. Objects and links that are being created or deleted by the application of the story pattern are labelled with the stereotypes «++» and «- -», respectively. The story pattern specifies the creation of a `member` link representing the RailCab's participation in the convoy operation simultaneously with its movement to the next track segment. The story pattern can be executed to transform the state graph into a new configuration if it contains a subgraph that *matches* the LHS of the story pattern.

Our modelling formalism also allows to express that certain objects or links are not permitted to appear in the current state graph. See for example the story pattern given in Figure 3. The crossed out `RailCab` object and the link connecting it to the `Convoy` object are not allowed to appear in the state graph. Such a restriction to the applicability of a story pattern is called a negative application condition (NAC).

The locking mechanism we developed is restricted to specific kinds of NACs. We use the terms *forbidden link* and *forbidden pair* to refer to these kinds of NACs. A forbidden link denotes a NAC that consists of a single link only. A forbidden pair denotes a NAC that consists of a single object and a link connecting this object to the LHS (as in Figure 3). A *connecting object* denotes an object within a LHS that is connected to a forbidden pair (e.g. the `Convoy` object in Figure 3).

In addition to the story patterns that define possible transformations, we need an initial configuration and a goal specification to feed the planning system with. A goal specification is a partly specified configuration that can be modelled as an ordinary object graph. Goal specifications are either generated from user input or predefined by the system designer. When the self-adapting system is in operation, initial configurations for the planning subsystem are generated from actual runtime states of the system.

3 Concurrent Execution

If we allow only sequential execution of reconfigurations, a plan is a sequence of graph transformations (or – in terms of PDDL – a sequence of ground actions). Considering reconfigurations as durative does not change this as long as we do not allow a concurrent execution of two reconfigurations. If we allow concurrency, the application intervals of two graph transformations can overlap and a plan thus is a set of tuples of points in time and

graph transformations. The questions that arise are: does the concurrent execution of two graph transformations result in any conflicts, and how can such a concurrent execution be avoided? An intuitive approach is to assume that the applicability check happens (in zero time) at the beginning of the reconfiguration and the actual change at its end. This is also assumed as default, i.e. when no further annotation is specified, in [TK11].

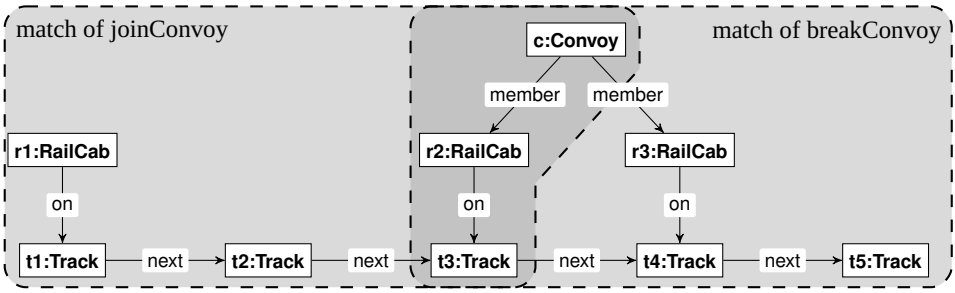

Figure 4: Configuration of the system in which two story patterns can be applied

Unfortunately, such an approach is not suitable for many situations. Consider for instance the state graph given in Figure 4 and the application of the story patterns `joinConvoy` and `breakConvoy` given in Figure 2 and 3. Each of the rules has only one match in the state graph. In Figure 4 they are highlighted by a dashed box. Let us assume that one of the reconfigurations, e.g. `breakConvoy`, is currently being applied. This means, its condition has already been checked but the alteration of the configuration has not yet been executed. The execution of a reconfiguration of RailCab `r1` joining the convoy makes no sense in this situation and should not be allowed because the convoy will be deinstantiated by `breakConvoy`. The problem is that the configuration is in the process of being changed, but this is not reflected in the intermediate state graph. Checking the applicability at the beginning of a reconfiguration and executing the alteration at its end is ineligible as a general solution. To solve this problem, we add information about the execution of the story patterns into the configuration. This can be seen as locking access to the elements of the state graph. Whether a second reconfiguration is allowed to be applied concurrently, can thus be checked by testing for the locks.

There are four cases of concurrency between two reconfigurations that might lead to conflicts. In all of them, an element is concurrently being read (required or forbidden) by a reconfiguration and being written (deleted or created) by another reconfiguration. They differ only in their beginning and ending times.

A schematic overview of the four conflict cases for the story patterns `joinConvoy` and `breakConvoy` is shown in Figure 5. The aforementioned example, in which the execution of `breakConvoy` begun before the `joinConvoy` was applied, corresponds to cases `a` and `b`. The reconfiguration removes the connection of RailCab `r2` to the convoy which is required by `joinConvoy`. In case `a`, `breakConvoy` ends first causing the convoy to be deinstantiated and the alteration of `joinConvoy` not to work anymore. In case `b`, the execution of `joinConvoy` ends first and the alteration of `breakConvoy` is still possible on the new state graph created by executing `joinConvoy`. However, such a concurrent

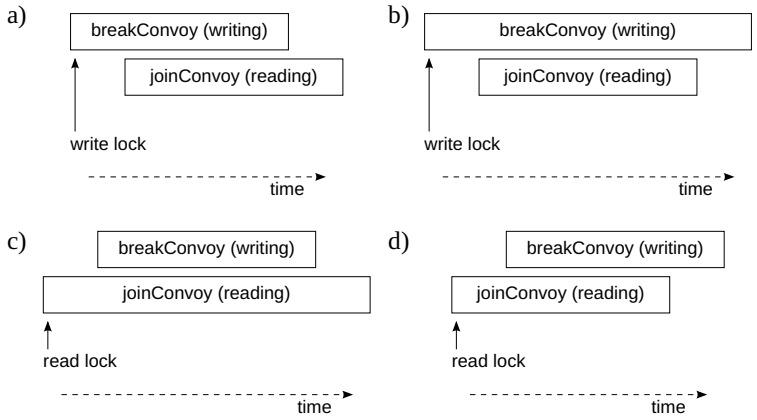

Figure 5: Four conflict cases that have to be solved by locking

execution of `joinConvoy` and `breakConvoy` is not intuitive: although `r1` joined a convoy and was not involved in the `breakConvoy` reconfiguration, there is no convoy that `r1` can be member of after the execution of both reconfigurations. From the perspective of `breakConvoy`, `joinConvoy` did not take the pending alteration of `breakConvoy` into account. Our solution to this problem encodes information about the deinstantiation of the convoy into the configuration by acquiring a write lock of the `Convoy` object when the `breakConvoy` reconfiguration starts and releasing the lock when the reconfiguration ends.

Although conflict cases `c` and `d` differ in their ordering of the reconfigurations' starting points, they are essentially caused by the same reason: either the alteration of `breakConvoy` causes the convoy to be deinstantiated while `r1` joins the convoy (case `c`) or `joinConvoy` does not take the pending alteration of `breakConvoy` into account (case `d`). These situations, however, require a different solution due to their different ordering of starting points. Since `joinConvoy` starts first, it cannot check the information about the deinstantiation of the convoy that `breakConvoy` is going to encode into the configuration by acquiring a write lock. Therefore, `joinConvoy` itself encodes into the configuration that it requires the `Convoy` object by acquiring a read lock of the `Convoy` object. Our solution approach essentially uses write locks on parts of the system's configuration to cope with cases `a` and `b` and read locks to cope with cases `c` and `d`.

4 Translation into PDDL

The planning technique we employ on our application scenario translates the story patterns into PDDL and uses an off-the-shelf planner to compute a reconfiguration plan that transforms a given initial configuration into a target configuration. We incorporated the locking mechanism explained in the last section directly into the translation, thus allowing for the computation of parallel plans that are guaranteed to be free of conflicts. The parallel plans

contain precise timing information for the application of the reconfigurations.

Since version 2.1 of PDDL [FL03] it is possible to specify durative actions that allow for concurrent execution. While PDDL's semantics already gives precise timing information for a resulting plan, its semantics regarding concurrent execution is too liberal from the perspective of our model. PDDL's semantics does not sufficiently constrain what kind of concurrent execution is allowed, thus burdening the designer of the planning domain with the complicated and error-prone definition of additional predicates to safely control whether a concurrent execution is allowed. Our translation scheme implements a suitable concurrency control by generating locking predicates in accordance with the locking mechanism we outline in the last section.

In PDDL, a planning task consists of a domain and a problem file. The domain defines action schemata, as well as types and predicates that can be used within action schemata. An action schema consists of a list of parameters, a precondition, and an effect. In the precondition, a list of literals that are required for applying the action can be specified. Similarly, the effect of an action specifies a list of literals that are obtained when the action is applied. An action is instantiated – in the context of PDDL this is called *grounding* – by substituting the parameters with objects that are defined in the problem file, thus transforming the first-order literals into a set of atomic facts that do not contain any free variables. In addition to the objects in the world, the problem file also defines the initial state and a goal specification, both in terms of a set of atomic facts.

Durative actions split the literals used in their precondition and effect into different sets according to their time of evaluation. Literals can be asserted `at_start`, `over_all`, and `at_end` when used in the precondition and be effective `at_start` and `at_end` when used in the effect. While `at_start` and `at_end` refer to the starting and ending timestamp of an action, `over_all` refers to the (open) interval during the action's execution.

The translation schema we describe here explains the construction of a PDDL domain file out of a given MECHATRONICUML model, i.e. a class diagram and a set of story patterns. Roughly spoken, the class diagram yields the declarations (of types and predicates) in the domain file and each story pattern yields an action schema. In doing so, the LHS of a story pattern constitutes the precondition of the action and the RHS its effect.

Class diagram. The translation process begins with the declaration of types, predicates, and functions. All declarations can be deduced from the class diagram. Listing 1 shows the generated declarations for our application scenario. Each type in the class diagram gives rise to a type in the type declaration section `:types`. Every type is a subtype of `Object`, the root of the type hierarchy. We use a predicate `active` for the supertype `Object` stating whether a given object exists, because PDDL does not allow for object creation or deletion. Each link in the class diagram is translated into a predicate that is parameterized by its source and target type. In PDDL, parameters are denoted by variable names followed by their types, e.g. `?track - Track`.

For now, we skip the declaration of the locks. We first cover the translation of story patterns by the example of `joinConvoy` without addressing the locking functionality, then give an overview of the modifications to realize the locks, and then stepwise extend the translation.

Listing 1: Generated declaration of types, predicates, and functions

```
(:types  Convoy - Object  RailCab - Object  Track - Object)
(:predicates
    (active  ?object - Object)
    (member_Convoy_RailCab  ?convoy - Convoy  ?railcab - RailCab)
    (on_RailCab_Track  ?railcab - RailCab  ?track - Track)
    (next_Track_Track  ?track1 - Track  ?track2 - Track)
)
```

Story patterns. Listing 2 shows the translation of the story pattern `joinConvoy`. All the conditions have to hold `at_start`, i.e. at the beginning of the story pattern's execution. The changes, i.e. the creation and deletion of objects or links, happen at the end of its execution.

Listing 2: Generated durative action `joinConvoy`

```
1  (:durative-action joinConvoy
2    :parameters (?c - Convoy  ?r1 - RailCab  ?r2 - RailCab  ?t1 - Track
3            ?t2 - Track  ?t3 - Track)
4    :duration (= ?duration 4)
5    :condition
6      (at start (and
7        (not (= ?r1 ?r2))  (not (= ?t1 ?t2))  (not (= ?t1 ?t3))
8        (not (= ?t2 ?t3))
9        (member_Convoy_RailCab ?c ?r2)
10       (on_RailCab_Track ?r1 ?t1)  (on_RailCab_Track ?r2 ?t3)
11       (next_Track_Track ?t1 ?t2)  (next_Track_Track ?t2 ?t3)
12       ...  % checking for locks
13     ))
14   :effect (and
15     (at start (and
16       ...  % locking
17     ))
18     (at end (and
19       (not (on_RailCab_Track ?r1 ?t1))
20       (member_Convoy_RailCab ?c ?r1)
21       (on_RailCab_Track ?r1 ?t2)
22       ...  % unlocking
23     ))
24   )
25 )
```

Every object in the story pattern – irrespective of whether an unchanged object or an object that is going to be created or deleted – is mapped to a parameter of the action. These parameters are checked for inequality in lines 7 and 8 because we employ injective matching in the graph transformation system.

Required links, i.e. unchanged links and links that are going to be deleted, cause the remaining literals in the condition of Listing 2. If there were forbidden links in the story pattern, they would have been translated into negative literals. The negative literal in the effect (line 19) is caused by the `on` link that is to be deleted and the two positive literals

(lines 20 and 21) by the `member` and `on` links that are to be created.

The translation of `joinConvoy` does not cover the case of a forbidden pair. While a forbidden link can simply be mapped into a negative literal, a forbidden pair, e.g. the NAC in the story pattern `breakConvoy`, has to be mapped into a negative existential quantification over the conjunction of object type and adjacent link that connects the object to the LHS. An isolated forbidden object, i.e. when no connecting object exists, can simply be mapped into a negative existential quantification involving only the `active` predicate. In both cases, inequality conditions are added if the object type has already been matched as a parameter, to be in accordance with employing injective matching. A more elaborate explanation of the mapping from non-durative graph transformation rules to PDDL action schemata that also covers attribute expressions is given in [TK11].

Locking functionality. Now we turn to the extensions of our translation scheme to integrate the locking mechanism we outlined in Section 3. First, the declaration of predicates and functions has to be extended to include the declaration of locks. Listing 3 shows the generated predicate and function declarations for the locks. For clarity, only declarations for the link between `Convoy` and `RailCab` are shown. The first six lines specify the read and write locks for ordinary objects and links. There is one pair of locks (read and write lock) for each object (lines 2 and 5) and one pair of locks for each link in the class diagram (lines 3 and 6). Write locks on objects and links are realized as predicates (exclusive lock, `true` means locked) because an object or link may not be accessed in any way if it is being deleted at the moment (or created in case of a forbidden link). As long as a reconfiguration does not manipulate an object or a link, a concurrent access is allowed. Therefore, read locks are realized as functions (shared lock, greater than 0 means locked).

Listing 3: Generated declaration of locks

```
1  % declaration of write locks for objects and links (in :predicates)
2    (writeNode_active  ?object - Object)
3    (writeEdge_member_Convoy_RailCab  ?convoy - Convoy  ?railcab - RailCab)
4  % declaration of read locks for objects and links (in :functions)
5    (readNode_active ?object - Object)
6    (readEdge_member_Convoy_RailCab  ?convoy - Convoy  ?railCab - RailCab)
7  % declaration of read locks for forbidden pairs (in :functions)
8    (readAdjacentToSource_member_Convoy_RailCab  ?convoy - Convoy)
9    (readAdjacentToTarget_member_Convoy_RailCab  ?railcab - RailCab)
10 % declaration of write locks for forbidden pairs (in :functions)
11   (writeAdjacentToSource_member_Convoy_RailCab  ?convoy - Convoy)
12   (writeAdjacentToTarget_member_Convoy_RailCab  ?railcab - RailCab)
```

The idea of the remaining locking predicates (lines 7–12) is more subtle. Objects within NACs cannot be locked via any of the aforementioned locking predicates (lines 1–6) because these objects do not exist in the current configuration, i.e. there is no explicit object in PDDL's propositional state that represents the object in the NAC. Instead, locking information is added to the objects they are connected to. This, of course, restricts our approach to specific kinds of NACs, namely forbidden links and forbidden pairs.

Locking of forbidden links (and links that are being created) is already supported via the locking predicates for links. Locking of forbidden pairs is supported by the functions in lines 8 and 9 which – pictorially speaking – add locking information to the connecting object. For each link predicate, there is a pair of locking predicates: the first (second) predicate is used to prevent the creation of a target (source) object that is connected to the source (target) object via a link of the same type and direction than the link within the forbidden pair. Objects that are being added are locked in a similar fashion via the functions in lines 11 and 12. However, for objects that are being added by a rule, locking information is attached to all object that the new objects are going to be connected to, i.e. for every appearing link that is connected to the appearing object a write lock for the pair of object and link is acquired. If there is no such object, then there is no need to lock anything since isolated objects do not interfere with any other object or link. Note, that the write locks are realized as functions instead of predicates because it is possible to apply more than one reconfiguration creating object connected to the same existing object concurrently. Also note, that while we support only two specific kinds of NACs, i.e. forbidden links and forbidden pairs, we support any kind of RHS.

Listing 4: Locking literals of `joinConvoy` to support (required) objects

```
1  % checking for locks (in :condition)
2    (not (writeNode_active ?c))  (not (writeNode_active ?r1))
3    (not (writeNode_active ?r2))  (not (writeNode_active ?t1))
4    (not (writeNode_active ?t2))  (not (writeNode_active ?t3))
5  % locking (in :effect, at start)
6    (increase (readNode_active ?c) 1)  (increase (readNode_active ?r1) 1)
7    (increase (readNode_active ?r2) 1)  (increase (readNode_active ?t1) 1)
8    (increase (readNode_active ?t2) 1)  (increase (readNode_active ?t3) 1)
9  % unlocking (in :effect, at end)
10   (decrease (readNode_active ?c) 1)  (decrease (readNode_active ?r1) 1)
11   (decrease (readNode_active ?r2) 1)  (decrease (readNode_active ?t1) 1)
12   (decrease (readNode_active ?t2) 1)  (decrease (readNode_active ?t3) 1)
```

We will now treat the generation of locking literals for the conditions and effects of action schemata and turn to the example of `joinConvoy` again. Listing 4 shows the locking literals generated to support the locking of required objects. For every positive literal that represents the existence of an object, a negative locking literal is added to the condition of the durative action (lines 2–4). As a result, the action is applicable only if none of the required objects (including objects being deleted) has been locked for writing. A shared read lock is acquired when the action is scheduled to begin (lines 6–8) and released when it ends (lines 10–12). The locking literals of required, i.e. unchanged and deleted, links are realized similarly. The same holds for forbidden links since they are translated into the same, yet negative literals as required links.

Although the story pattern `joinConvoy` does not contain forbidden pairs itself, it generates locking literals for the support of forbidden pairs. This is necessary to avoid conflicts between forbidden pairs and the creation of links: a forbidden pair might interfere with creating a link because the link could be adjacent to an object of the same type as the object in the forbidden pair. To avoid this, further locking literals are generated for every

Listing 5: Locking literals of `joinConvoy` to support forbidden pairs

```
1  % checking for locks (in :condition)
2    (= (readAdjacentToSource_member_Convoy_RailCab ?c) 0)   % link is being created
3    (= (readAdjacentToTarget_member_Convoy_RailCab ?r1) 0)  % link is being created
4    (= (readAdjacentToSource_on_RailCab_Track ?r1) 0)       % link is being created
5    (= (readAdjacentToTarget_on_RailCab_Track ?t2) 0)       % link is being created
6  % locking (in :effect, at start)
7    (increase (writeAdjacentToSource_member_Convoy_RailCab ?c))   % link is b. created
8    (increase (writeAdjacentToTarget_member_Convoy_RailCab ?r1))  % link is b. created
9    (increase (writeAdjacentToSource_on_RailCab_Track ?r1))       % link is b. created
10   (increase (writeAdjacentToTarget_on_RailCab_Track ?t2))       % link is b. created
11 % unlocking (in :effect, at end)
12   ...  % analogue
```

appearing link. Listing 5 shows these locking literals. The locking literals in lines 2–5 state that no forbidden pair lock may be acquired for any of the objects that are adjacent to the appearing links. Such a lock will only be acquired if the story pattern of another reconfiguration that is applied concurrently has a forbidden pair connected to one of these objects. The locking literals in the effect (lines 7–10) itself acquire write locks of forbidden pairs to guarantee that no concurrent reconfiguration with a forbidden pair connected to one of these objects will be applied concurrently. Reconfigurations with forbidden pairs check for these write locks in their condition.

Implementation and application. The proposed translation scheme was implemented for the FUJABA TOOL SUITE using *Xpand*, a code generation language for EMF models. Running the translator on a given model, i.e. a class diagram and a set of story patterns, generates the corresponding PDDL planning domain.

Our application scenario includes reconfigurations to move RailCabs or convoys of RailCabs and reconfigurations related to convoy de-/instantiation and membership change. All these reconfigurations are available in the generated planning domain. The planning problems associated with the generated domain consist of 30 track segments, 2–4 RailCabs, a few junctions and specify initial and target track segments for the RailCabs. Such problems can be solved by SGPlan_6 within a few seconds: our planning tasks took approx. 1.4, 3.8, and 10.1 seconds (involving 2, 3, or 4 RailCabs, resp.) on an Intel Core i7-2640M running at 2.8GHz with 8GB RAM. The generated reconfiguration plans take advantage of parallel execution of actions when possible, while guaranteeing that concurrently executed actions do not interfere with each other. With regard to the application scenario, this means that RailCabs operate in parallel if they are sufficiently apart from each other, but wait for the execution of other RailCabs' reconfigurations if necessary, e.g. to clear a common track segment.

Listing 6 shows an excerpt of a resulting plan for the problem instance involving 4 RailCabs. During the interval [60–64], `railcab2` and `railcab3` operate in convoy mode. From 64 to 68, they break up the convoy operation because the underlying domain specifies a Y junction between `track19`, `track20`, and `track25`, and they need to move along

different routes to arrive at their target locations. To do so, `railcab2` has to fall back, i.e. it still occupies `track19` at 68. Concurrently, i.e. during the interval [60–68], `railcab0` moves from `track16` to `track18` but waits from 68 to 72 to not crash into `railcab2`.

Listing 6: Excerpt of a resulting plan for 4 RailCabs

```
60.041: (MOVE railcab0 track16 track17) [4.0000]
60.042: (MOVECONVOY convoy0 railcab2 railcab3 track18 track19 track20) [4.0000]
64.043: (BREAKCONVOY convoy0 railcab2 railcab3 track19 track20 track21) [4.0000]
64.044: (MOVE railcab0 track17 track18) [4.0000]
68.045: (MOVE railcab3 track21 track22) [4.0000]
68.046: (CREATECONVOY convoy0 railcab2 railcab1 track19 track25 track26) [4.0000]
72.047: (MOVE railcab3 track22 track23) [4.0000]
72.048: (MOVECONVOY convoy0 railcab2 railcab1 track25 track26 track27) [4.0000]
72.049: (MOVE railcab0 track18 track19) [4.0000]
```

5 Related Work

While planning and scheduling is a discipline in artificial intelligence research that has made many advances in the last decades, only few moves have been made to tie planning techniques with the software engineering domain, e.g. [VSF+09] and [SKM07]. The most promising approaches rely on graph transformation systems as an underlying formalism to specify the planning tasks because of their intuitive representation and close association to model-based software engineering. An early attempt into this direction came from Edelkamp and Rensink [ER07]. They showed manual translations from planning tasks specified with graph transformation rules into PDDL and identified some advantages of planning directly on graph transformation systems: the possibility to reduce the state space by representing isomorphic graphs only once and the support for dynamic object creation and deletion. These advantages gave reason for techniques that directly use graph transformation systems for planning, like [RW10] and [EW11]. In [RW10] the planning task is solved by transforming it into a model checking problem, in [EW11] by using heuristic search techniques. None of these approaches support time-consuming reconfigurations.

Tichy and Klöpper [TK11] were first to present an automatic translation of graph transformation rules into PDDL actions. The support for time-consuming reconfigurations was addressed in terms of stereotypes; concurrency issues were not treated. Meijer [Mei12] also provides a translation between graph transformations and PDDL but does not cover time or durative actions. The developed translator works in both directions, i.e. planning tasks formulated in PDDL can also be translated back into a graph transformation system. As opposed to our technique, the main focus of [Mei12] lies on the backward direction. The employed graph transformation tool, however, needs to support existential quantification on edges to match the semantics of PDDL[2].

[2] In PDDL, a literal that is going to be deleted by an action does not have to be present if it is not required in the precondition. In such a case the action is still applicable but does not change the literal.

6 Conclusion and Future Work

We presented a model-based approach for planning time-consuming architectural reconfigurations for self-adaptive systems. Our technique solves planning tasks for graph transformation systems by translating them into an input for efficient off-the-shelf planning systems. It computes temporal plans where each reconfiguration step has its own associated duration and reconfiguration steps can be carried out in parallel. Unintended interferences, e.g. the deinstantiation and use of a software component at the same time, are avoided thanks to the locking mechanism that we integrated into the translation scheme.

Currently, we investigate the use of domain-independent heuristics for planning techniques that are applied directly on graph transformation systems as an alternative approach. We plan to do a detailed evaluation comparing the efficiency of planning directly on graph transformation systems with the efficiency of running off-the-shelf planning systems on corresponding PDDL models generated by our translator.

As for our modelling approach and its locking mechanism, we plan to extend our approach with *required concurrency*, cf. [CKMW07]: we should be able to specify a reconfiguration that explicitly requires the concurrent application of another reconfiguration. Currently, locks set by a reconfiguration can only *restrict* other reconfigurations from being applicable (which is why we called them locks), instead of *enabling* their applicability (like a window of opportunity for another reconfiguration).

References

[BBB+12] S. Becker, C. Brenner, C. Brink, S. Dziwok, R. Löffler, C. Heinzemann, U. Pohlmann, W. Schäfer, J. Suck, and O. Sudmann. The MechatronicUML Design Method – Process, Syntax, and Semantics. Technical report, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, 2012.

[BBG+06] Basil Becker, Dirk Beyer, Holger Giese, Florian Klein, and Daniela Schilling. Symbolic Invariant Verification for Systems with Dynamic Structural Adaptation. In *28th Int. Conf. on Software Engineering (ICSE 2006)*. ACM Press, May 2006.

[BCDW04] Jeremy S. Bradbury, James R. Cordy, Juergen Dingel, and Michel Wermelinger. A Survey of Self-Management in Dynamic Software Architecture Specification. In *ACM SIGSOFT Workshop on Self-Managed Systems (WOSS 2004)*, 2004.

[CKMW07] William Cushing, Subbarao Kambhampati, Mausam, and Daniel S. Weld. When is Temporal Planning Really Temporal? In *20th Int. Joint Conf. on Artificial Intelligence (IJCAI 2007)*, pages 1852–1859. Morgan Kaufmann Publishers Inc., 2007.

[CWH06] Yixin Chen, Benjamin W. Wah, and Chih-Wei Hsu. Temporal Planning using Subgoal Partitioning and Resolution in SGPlan. *J. Artif. Intell. Res. (JAIR)*, 26:323–369, 2006.

[EHK+97] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic Approaches to Graph Transformation II: Single Pushout Approach and Comparison with Double Pushout Approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*, chapter 4, pages 247–312. World Scientific, 1997.

[ER07] Stefan Edelkamp and Arend Rensink. Graph Transformation and AI Planning. In *Int. Competition on Knowledge Engineering for Planning and Scheduling (ICKEPS 2007)*, September 2007.

[EW11] H.-Christian Estler and Heike Wehrheim. Heuristic Search-Based Planning for Graph Transformation Systems. In *Workshop on Knowledge Engineering for Planning and Scheduling (KEPS 2011)*, pages 54–61, 2011.

[FL03] Maria Fox and Derek Long. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research (JAIR)*, 20:61–124, 2003.

[FNTZ98] Thorsten Fischer, Jörg Niere, Lars Torunski, and Albert Zündorf. Story Diagrams: A New Graph Rewrite Language based on the Unified Modeling Language. In *6th Int. Workshop on Theory and Application of Graph Transformations (TAGT 1998)*, 1998.

[GMK02] Ioannis Georgiadis, Jeff Magee, and Jeff Kramer. Self-organising Software Architectures for Distributed Systems. In *Workshop on Self-Healing Systems (WOSS 2002)*, pages 33–38, 2002.

[KM07] Jeff Kramer and Jeff Magee. Self-Managed Systems: an Architectural Challenge. In *Future of Software Engineering (FOSE 2007)*, pages 259–268, Washington, DC, USA, 2007. IEEE Computer Society.

[LM98] Daniel Le Métayer. Describing Software Architecture Styles Using Graph Grammars. *IEEE Transactions on Software Engineering*, 24(7):521–533, 1998.

[Mei12] Ronald Meijer. PDDL Planning Problems and GROOVE Graph Transformations: Combining Two Worlds with a Translator. In *17th Twente Student Conference on IT*, June 2012.

[RW10] Malte Röhs and Heike Wehrheim. Sichere Konfigurationsplanung selbst-adaptierender Systeme durch Model Checking. In J. Gausemeier, F. Rammig, W. Schäfer, and A. Trächtler, editors, *Entwurf mechatronischer Systeme*, volume 272 of *HNI-Verlagsschriftenreihe*, pages 253–265. Heinz Nixdorf Institut, 2010.

[SKM07] Ron M. Simpson, Diane E. Kitchin, and T. L. McCluskey. Planning Domain Definition Using GIPO. *Knowledge Engineering Review*, 22(2):117–134, June 2007.

[SWW11] Dominik Steenken, Heike Wehrheim, and Daniel Wonisch. Sound and Complete Abstract Graph Transformation. In *14th Brazilian Symposium on Formal Methods (SBMF 2011)*, pages 92–107, September 2011.

[THHO08] Matthias Tichy, Stefan Henkler, Jörg Holtmann, and Simon Oberthür. Component Story Diagrams: A Transformation Language for Component Structures in Mechatronic Systems. In *4th Workshop on Object-oriented Modeling of Embedded Real-Time Systems (OMER 4)*, pages 27–38, 2008.

[TK11] Matthias Tichy and Benjamin Klöpper. Planning Self-Adaptation with Graph Transformations. In Andy Schürr, Dániel Varró, and Gergely Varró, editors, *Int. Symp. on Applications of Graph Transformation with Industrial Relevance (AGTIVE 2011)*, volume 7233 of *LNCS*. Springer Verlag, 2011.

[VSF+09] Tiago Stegun Vaquero, José Reinalda Silva, Marcer Ferreira, Flavio Tonidandel, and J. Christopher Beck. From Requirements and Analysis to PDDL in itSIMPLE3.0. In *19th Int. Conf. on Automated Planning and Scheduling (ICAPS 2009)*, 2009.

[WF02] Michel Wermelinger and José Luiz Fiadeiro. A Graph Transformation Approach to Software Architecture Reconfiguration. *Science of Computer Programming*, 44(2):133–155, August 2002.