Planning Self-adaption with Graph Transformations

Matthias Tichy[1] and Benjamin Klöpper[2]

[1] Software Engineering Division,
Chalmers University of Technology and University of Gothenburg,
Gothenburg, Sweden
tichy@chalmers.se
[2] National Institute of Informatics (NII), Tokyo, Japan
kloepper@nii.co.jp

Abstract. Self-adaptive systems autonomously adjust their behavior in order to achieve their goals despite changes in the environment and the system itself. Self-adaption is typically implemented in software and often expressed in terms of architectural reconfiguration. The graph transformation formalism is a natural way to model the architectural reconfiguration in self-adaptive systems. In this paper, we present (1) how we employ graph transformations for the specification of architectural reconfiguration and (2) how we transform graph transformations into actions of the Planning Domain Definition Language (PDDL) in order to use off-the-shelf tools for the computation of self-adaptation plans. We illustrate our approach by a self-healing process and show the results of a simulation case study.

Keywords: Self-adaptive systems, graph transformations, planning, PDDL.

1 Introduction

The complexity of today's systems enforces that more and more decisions are taken by the system itself. This is resembled by the current trend to systems which exhibit self-x properties like self-healing, self-optimizing, self-adaption. Self-x properties cause additional complexity and dynamics within the system. Therefore, appropriate development approaches have to be employed. The architecture is one of the key issues in building self-x systems [19,17]. In particular, self-adaptation can be realized by adapting the architectural configuration by adding and removing components as well as replacing them.

Kramer and Magee [17] presented a three-layer architecture for self-managed systems consisting of the following layers: (1) *goal management*, (2) *change management*, and (3) *component control*. The component control layer contains the architectural configuration of the self-adaptive system, i.e., the components and their connections which are active in a certain state. Besides the execution of the components, this layer is responsible for the execution of reconfiguration

A. Schürr, D. Varró, and G. Varró (Eds.): AGTIVE 2011, LNCS 7233, pp. 137–152, 2012.
© Springer-Verlag Berlin Heidelberg 2012

plans. These plans, which describe the orderly adding, removing, and replacing of components and connectors, are executed to transform the current configuration into a new one in reaction to a new situation or event. They are stored in the change management layer and computed by the goal management layer.

Several approaches [25,18,28,4,26] employ graph transformations for the specification of architectural reconfiguration of self-adaptive systems. Graph transformations enable the application of formal verification approaches (e.g., [21,3]) and code generation [9] for execution during runtime by providing a sound formal basis. However, all of these approaches only address the modeling aspect of reconfiguration and do not address the computation of reconfiguration plans to meet the goals.

In this paper, we present how graph transformations can be integrated with automated planning approaches [12] to compute reconfiguration plans. We model the system structure using class diagrams and employ story patterns [9,29] as specific graph transformation formalism. Additionally, we extend story patterns by modeling elements for temporal properties to enable temporal planning. Similar to [6], we translate these models to the Planning Domain Definition Language [10] to enable the application of off-the-shelf planning software like SGPlan [5].

In the next section, we introduce the running example, which is about self-healing as a special case of self-adaptation, which is used to illustrate our approach. Section 3 gives a short introduction how we model the structure and the self-adaption behavior of our running example. The translation of the models to the planning domain definition language is described in section 4. Thereafter, we present an extension of our approach to durative actions and temporal planning in Section 5. In Section 6, we present results of simulation experiments for our self-healing application scenario. After a discussion of related work in Section 7, we conclude with an outlook on future work in Section 8.

2 Example

As an application example, we consider the self-healing process in an automotive application as presented in [16]. While it is currently not the case, we assume that, in the future, it is possible that software components can be deployed, started and stopped on electronic control units (ECU) at runtime. Currently, the deployment of software components to ECUs is done at design time but online reconfiguration gains interest and will eventually be realized. The AUTOSAR standard [11] with its standardized interfaces and the run-time environment (RTE) is the first step towards a system that can be reconfigured.

Our self-healing process reacts to failures of software components and hardware nodes (ECUs) by, for example, starting failed software components on working nodes, moving software components from a source to a target node, disconnecting and reconnecting software components. While the original self-healing process [16] considers redundant software components, we do not consider redundancy in this paper in order to keep the examples smaller.

Figure 1 shows an example of our self-healing process. On the left hand side of that figure, four nodes are shown which execute five component instances. `node1` has experienced a failure and, thus, component `c1` is not working anymore.

The self-healing process now reacts to this failure by computation and subsequently execution of a self-healing plan. This self-healing plan is comprised of the actions transfer which transfers the code of a component to a node, the action createInstance which instantiates the component of a node, and destroy which destroys a component instance on a node. For this example, the plan basically results in moving component c2 from node2 to node3 in order to free up space to subsequently instantiate the failed component c1 on node node2.

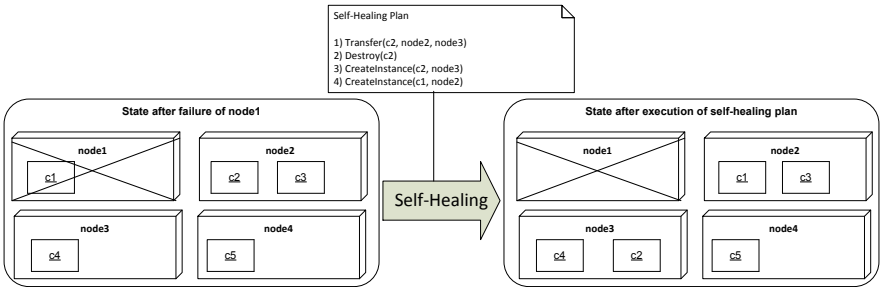

Fig. 1. Self-healing a failure of node1

A good example of a safety-relevant automotive subsystem is an adaptive cruise control (ACC). An ACC is an advanced tempomat, its functionality is to accelerate the car to the driver specified velocity, if no obstacle is detected. If an obstacle is detected, the car is first decelerated and then controls the gap between the car and the obstacle (mostly another car). The *adaptive* in its name comes from this change of behavior. Figure 2 shows the software components of a sample adaptive cruise control system. In this paper, we do not specifically target the self-healing of this adaptive cruise control system but address the general case of self-healing component-based systems as a running example.

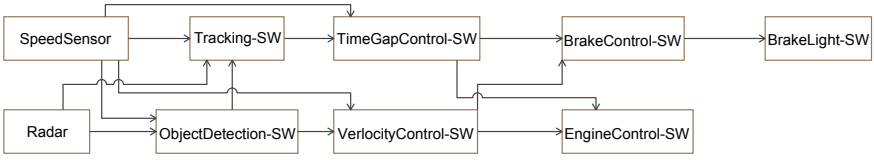

Fig. 2. Software components of an adaptive cruise control system [16]

3 Modeling with Graph Transformations

We employ the story pattern [9,29] graph transformation formalism, which is tightly integrated with the UML. It employs class diagrams for the specification of structure (similar to typed graphs in graph transformations) and refined collaboration diagrams for the specification of a graph transformation.

3.1 Specification of Structure

Figure 3 shows the class diagram for our self-healing scenario. Each component represents a software component. Pairs of components may have to communicate with each other. Nodes represent the computation hardware which are used to execute the software components. Before a component can be started on a node, the software code of the component has to be deployed to that specific node. Nodes are connected to other nodes. Components which communicate with each other must be executed either on the same node or on connected nodes. Each connection provides a certain transfer rate.

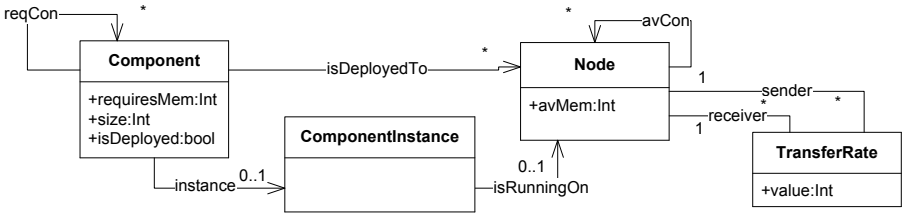

Fig. 3. Class diagram modeling the structure of the self-healing system

For the subsequent translation to PDDL, we require that for each class the maximum number of instances is specified by the developer. For our example, this maximum number of instances is known during design-time as the number of component types is known as well as the number of nodes. The number of component instances is equal to the number of component types as we specifically choose to have only a single instance of each component type in the system. Thus, this requirement is feasible for our scenario. This requirement is also typical in embedded systems. However, in other applications or domains this requirement might not hold.

3.2 Specification of Self-adaption Actions

The reconfiguration actions are specified with story patterns which are typed over the class diagram presented in Figure 3, i.e., they transform instances of this class diagram.

Story patterns follow the single pushout [22] formalism. The left hand side and the right hand side are merged into a single graph in story patterns. In this graph, all nodes and edges which are in the left hand side but not in the right hand side are marked as `delete`. These nodes and edges are deleted by the execution of the rule. All nodes and edges which are in the right hand side but not in the left hand side are marked as `create`. They are created by the execution of the rule. Simple negative application conditions can be modeled by appropriately annotating edges and nodes in the diagram. It is not allowed that negative nodes are attached to negative edges [29]. In contrast to standard story patterns, we do not support bound objects.

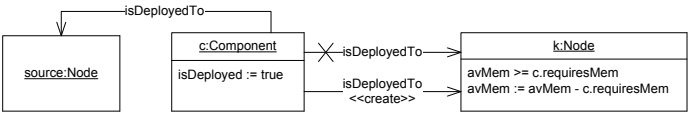

Fig. 4. Story pattern which specifies the deployment of a component c to node k

Figure 4 shows a story pattern which specifies the transfer of the code of a component c from a source node source to a target node k to enable starting the component on that node. The fact that the code of a component is available on a node is modeled as the existence of an isDeployedTo link between the component and the node. Additionally, the story pattern specifies that the amount of available memory on node k must be greater than the required memory of component c prior to execution. After execution the available memory is reduced.

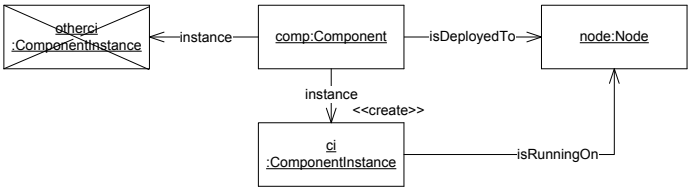

Fig. 5. Story pattern which specifies the instantiation of a component on a node

Figure 5 shows the story pattern concerning the instantiation of a component on a node. This story pattern can be executed if (1) the component's code has been previously deployed to the node and (2) the component is not already instantiated anywhere in the system. The first condition is expressed by the link isDeployedTo between comp and node. The second condition is expressed by the negative object otherci.

3.3 Specification of Goals

Finally, the goal has to be modeled for the self-healing system. Using only the left hand side of story patterns as in [8], it is possible to model a concrete situation which shall be reached by the computed plan. Though, this is rather cumbersome, especially for a large number of objects. Instead we use enhanced story patterns [24] which extend story patterns with subpatterns and quantifiers. Figure 6 shows the specification of the goal that for every component there should exist a component instance which is running on a node.

We do not provide modeling support for the initial state as the initial state is simply the current state of the self-adaptive system during runtime.

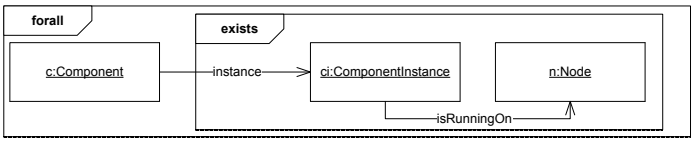

Fig. 6. Enhanced story pattern modeling the goal

4 Translation to PDDL

There exist several different representations for planning problems [12]. Set-theoretic approaches represent states by a set of propositions. Each action specifies which propositions have to hold in the state for the action to be applicable. Additionally, an action specifies which propositions will be added and removed to create a new state. The classical representation uses first-order literals and logical operators instead of propositions.

The Planning Domain Definition Language (PDDL) has been developed as a standard language for planning problems in order to compare different tools and algorithms. In order to be applicable for a wide variety of planning problems and tools, several extensions with respect to the classical representation have been added to the PDDL, e.g., typing, durative actions for temporal planning, fluents for representation of numerical values.

A planning problem in PDDL consists of two parts – the domain and the problem. The domain defines the types, the predicates, the functions, and the actions. A PDDL action is defined by a name, a list of parameters, a precondition and an effect. All objects which are referred to in the precondition and the effect have to be included in the list of parameters. The action can only be executed if the precondition is satisfied. The effect holds after the execution of the action. The problem defines the objects as well as the initial state and the goal state. The planning system then tries to compute a sequence of actions (including the arguments for the action parameters) which transforms the initial state into the goal state.

The PDDL extension `:typing` enables the modeling of types and generalization in the planning model. Thus, classes are translated to types in the domain definition; the associations are translated to predicates over the types. Story patterns are translated to actions in the planning domain. The left hand is translated to the precondition and the right hand side to the effect. Although the elements of both models map well to each other, there are details to consider.

4.1 Types, Predicates and Functions

Basically, classes are translated to types. Generalizations in the class diagrams are translated as well. Listing 1 shows the types generated from the class diagram of Figure 3. All types extend the general predefined type `object` denoted by the suffix "`- object`".

Listing 1: Mapping of classes to types in the PDDL

```
1 (:types
2    Component Node TransferRate ComponentInstance - object
3 )
```

Associations are translated to predicates over the source and target types. We support unidirectional and bidirectional associations. For example, the unidirectional association isDeployedTo between Component and Node is translated to the predicate (isDeployedTo ?component - Component ?node - Node). Listing 2 shows the predicates which are generated from the associations of the class diagram. For bidirectional associations, only one direction is translated to predicates in order to minimize the planning domain as bidirectional navigability is already provided by an unidirectional reference in PDDL.

Listing 2: Translation of associations to predicates

```
1  (:predicates
2     (exist ?object - object)
3     (isRunningOn ?component - Component ?node - Node)
4     (reqConn ?component - Component ?component - Component)
5     (instance ?component - Component ?componentinstance - ComponentInstance)
6     (isDeployedTo ?component - Component ?node - Node)
7     (avConnections ?node - Node ?node - Node)
8     (sourceTransferRate ?node - Node ?transferrate - TransferRate)
9     (targetTransferRate ?node - Node ?transferrate - TransferRate)
10    (runningOn ?componentinstance - ComponentInstance ?node - Node)
11    (isDeployed ?component - Component)
12 )
```

The PDDL prohibits the creation and deletion of objects to preserve a finite state space. As node creation and deletion is an important feature of graph transformations, we decided to emulate object creation and deletion by using the special predicate (exist ?object - object). We require a fixed number of objects of each type in the initial state of the planning problem. We emulate object creation and deletion by setting the exist predicate appropriately. Finally, Boolean attributes are also translated to predicates, e.g., the attribute `isDeployed`.

Functions provide mappings from object tuples to the realm of real numbers. This enables the translation of all numerical attributes from class diagrams. For example, the function (avMem ?n - Node) stores the amount of that node's memory which is increased and reduced depending on the number of components who are instantiated on a given node.

4.2 Actions

In the following, we present how story patterns are translated to PDDL actions. The preconditions and the effects of PDDL actions mirror naturally the left hand side and right hand side of a graph transformation. Listing 4 shows the PDDL action for the story pattern from Figure 4. We translate all nodes of the story

Listing 3: Translation of integer attributes to functions

```
1 (:functions
2   (requiresMem ?component - Component)
3   (size ?component - Component)
4   (avMem ?node - Node)
5   (value ?transferrate - TransferRate)
6 )
```

pattern to parameters of the action. The planner will bind these parameters to objects in such a way that the precondition is satisfied. The action transfer has three parameters for the two nodes source and k and the component c. The

Listing 4: Transfer of component code from source to target node

```
1  (:action transfer
2    :parameters (
3      ?source - Node ?k - Node ?c - Component
4    )
5    :precondition (and
6      (exist ?source) (exist ?k) (exist ?c)
7      (not (= ?source ?k))
8      (isDeployedTo ?c ?source)
9      (not (isDeployedTo ?c ?k))
10     (>= (avMem ?k) (requiresMem ?c))
11   )
12   :effect (and
13     (isDeployedTo ?c ?k)
14     (isDeployed ?c)
15     (decrease (avMem ?k) (requiresMem ?c))
16   )
17 )
```

precondition requires that all bound objects are indeed existing (line 6). The story pattern formalism uses a graph isomorphism, i.e., that two nodes of the graph transformation cannot be bound to the same node in the host graph. Consequently, we check in line 7 that the two nodes source and k are different. We translate the isDeployedTo-edge to the predicate in line 8.

The effect of the action simply states that the isDeployedTo predicate holds for the component c and the node k as the semantics of PDDL effects are that everything remains unchanged except the explicitly stated effect.

Attribute Expressions. Arithmetic expressions are supported by PDDL functions. The precondition can contain comparisons concerning functions. Values are assigned to functions in the effect. Concerning the transfer action, the precondition checks whether the available memory is greater or equal than the memory required by the component in line 10. In line 15, the available memory is decreased by this required amount of memory. Finally, the assignment of the Boolean variable isDeployed is part of the effect as well (line 14).

Negative Nodes and Edges. Story patterns enable the specification of simple negative application conditions by annotating nodes and edges that the node or the edge must not match.

The story pattern of Figure 4 contains a negative edge isDeployedTo between component c and node k. Thus, the story pattern is only applicable if the component c has not been already deployed to the node k. This is translated to a negated predicate as shown in line 9.

The case of a negative node is more complex. The semantics of a negative node [29] is that the matching of the left hand side minus the negative node must not be extendable to include a matching of the negative node as well. This is translated to a negative existential quantification over the objects of this type including the edges connected to the negative node as in lines 8 to 10 of Listing 5 which is the PDDL translation of the story pattern shown in Figure 5.

Again, special care has to be taken concerning injective matching. If a node is already positively matched, it will be excluded from the negative existential quantification.

Object Creation and Deletion. There exist several possibilities to emulate object creation and deletion. Naively, objects can be emulated by predicates which are set to true and false accordingly. This does only work for the case that it is not required that an object is identifiable. This is typically not suitable for graph transformations. As mentioned earlier, we decided to allocate a fixed number of objects of each class and use the additional predicate exist to denote whether the object exists or not.

Listing 5 shows the PDDL translation of Figure 5. Similar to Listing 4, we initially check for all nodes which are in the left hand side of the story pattern whether they exist in line 6. The object which will be created by the story pattern must not exist prior to the execution with the predicate in line 7. It is created in the effect (line 16).

Listing 5: Creating objects

```
1  (:action createInstance
2    :parameters (
3      ?comp - Component ?node - Node ?ci - ComponentInstance
4    )
5    :precondition (and
6      (exist ?comp) (exist ?node)
7      (not (exist ?ci))
8      (not (exists (?otherci - ComponentInstance)
9        (instance ?comp ?otherci)
10     ))
11     (not (instance ?comp ?ci)) (not (runningOn ?ci ?node))
12   )
13   :effect (and
14     (instance ?comp ?ci)
15     (runningOn ?ci ?node)
16     (exist ?ci)
17   )
18 )
```

Listing 6 shows how objects are destroyed by an action. The story pattern formalism follows the single pushout approach [22]. Therefore, we do not require that the dangling condition is satisfied and simply delete all edges related to the node (lines 6 and 7). The class diagram (see Figure 3) holds the information which edges we have to remove when destroying an object.

Listing 6: Destroying objects

```
1 (:action destroy
2   :parameters (?ci - ComponentInstance))
3   :precondition (and (exist ?ci))
4   :effect (and
5     (not (exist ?ci))
6     (forall (?o - Node) (not (runningOn ?ci ?o)))
7     (forall (?o - Component) (not (instance ?o ?ci)))
8   )
9 )
```

The model of the goal state shown in Figure 6 is translated to the PDDL in a similar way as the left hand side of story pattern with appropriate handling of the quantification.

5 Adding Temporal Properties

PDDL 2.1 [10] introduced syntax and semantics for temporal planning. Temporal planning relaxes the assumption of classical planning that events and actions have no duration. This abstraction is often not suitable as in reality actions *do* occur over a time span. Therefore, durations can be annotated to actions in temporal planning. As this allows concurrent actions, preconditions and effects have to be annotated. Three different temporal annotations are supported which can be combined: (1) at start, the precondition has to be satisfied at the beginning of the action, (2) at end, the precondition has to be satisfied at the end of the action, and (3), over all, the precondition has to be satisfied during the action. Effects have to be annotated with at start or at end.

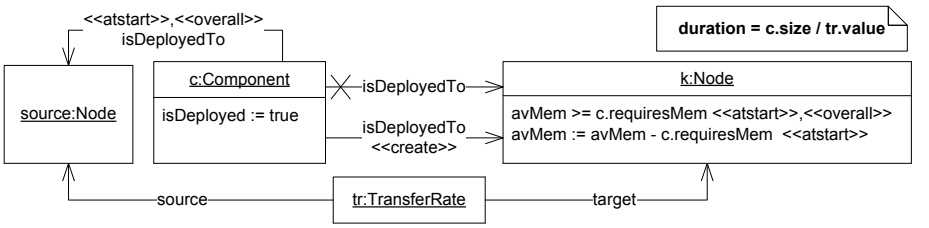

Fig. 7. Story pattern which specifies the deployment of a component c to node k including a duration

In general, story patterns do not consider time. Timed story patterns [14] are only concerned with *when* the pattern is executed, but not about the duration of its execution. We extend the story pattern by a duration fragment, which is used for the specification of the duration. Figure 7 shows this extension. The duration is computed based on the component size and the transfer rate for the connection between the nodes in our example. We annotate elements of the story pattern with the stereotypes ≪atstart≫, ≪atend≫, ≪overall≫ to specify the required temporal properties.

Listing 7: durative-action transfer

```
1  (:durative-action transfer
2    :parameters (?c - component ?k - Node ?source - Node ?tr - TransferRate)
3    :duration (= ?duration (/ (size ?c) (value ?tr)))
4    :condition (and
5      ...
6      (over all (isDeployedTo ?c ?source))
7      (at start (isDeployedTo ?c ?source))
8      (at start (sourceTransferRate ?rate ?source))
9      (at start (targetTransferRate ?rate ?target))
10     ...
11   )
12   :effect (and
13     (at end (isDeployedTo ?c ?k))
14     (at end (isDeployed ?c ))
15     (at start (decrease (avMem ?k) (requiresMem ?c)))
16   )
17 )
```

In contrast to the PDDL, we assume the following defaults in the case that the developer does not specify temporal stereotypes for the sake of visual clarity. All elements of the left hand side are assumed to have the stereotype ≪atstart≫ whereas all effects are assumed to have the stereotype ≪atend≫.

Listing 7 shows an excerpt from the durative action generated from the story pattern of Figure 7. The specification of the duration is shown in line 3. During the whole execution of the action, the component c must be deployed to the source node. As the temporal plan can schedule actions in parallel, we require that at the beginning of the action the available memory of node k must already be decreased by the required amount of component c.

6 Simulation Experiments

In order to show the feasibility of our approach, we conducted simulation experiments for the self-healing scenario. The scenario has been extended by resources which are required by components and provided by nodes, communication buses between the nodes as well as redundant allocation of components to nodes. The discrete event-based simulation environment simulates (1) failures of nodes, (2) repairs of nodes, and (3) periodic self-healing activities which are comprised of computing and executing self-healing plans. We abstract from the actual behavior of the components and restrict the simulation to the failures

and the self-healing process. The plans are computed by the SGPlan automated planning software. The simulated system consists of 12 component types and 5 nodes connected by two communication buses. Node failures are randomly distributed by a negative exponential distribution $f_\lambda(x) = \lambda e^{-\lambda x}$ with a failure rate of $\lambda = 0.0001$. Every 5 time units, the self-healing part of the system checks whether any component type is not instantiated. In that case the planner is called and the resulting plan is executed.

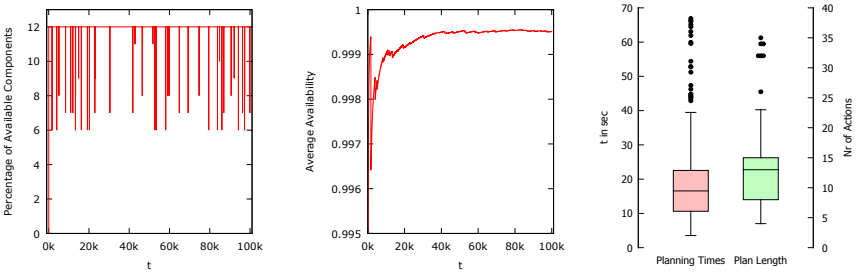

Fig. 8. Results from the simulation experiments

On the left side of Figure 8, the number of available component instances at each point of a single simulation run for 100.000 time units is shown. The complete system is available if each component type is instantiated on a node, i.e., if 12 component instances are available. At 53 points in time, a node fails which results in the failure of the components which are instantiated on that node. 10 of that 53 failures did happen to nodes which had no component types instantiated and, thus, resulted in no reduction of the number of component instances. After computing and executing the self-healing plan in reaction to a node failure, the number of available component instances increases to 12 again.

In the middle of Figure 8, the average availability of the system during the same simulation run is shown. We define availability as the probability that all component types of the system are instantiated at a certain point of time. The system starts with no instantiated components. Consequently, the average availability rises at the start of the simulation and reacts heavily to node failures. At the end of the simulation run, it is stable at 0.9995.

On the right side of that figure, we report the time taken by the planner to compute the plan as well as the planning length based on 277 calls to the planner. The planner is executed on an Intel Core2Duo with two cores at 2,53 GHz and 4 GB of RAM. Though, the planner uses only one core.

7 Related Work

In [4], a model-driven approach for self-adaption has been presented which also applied graph transformations for the specification of component reconfigurations. The graph transformations are used to specify goals, but the approach

supports only the monitoring of these goals and not the computation of reconfiguration plans to achieve them. In general, planning is an important method in self-adapting and self-configuring systems. For instance, Arshad et al. [2] introduced a PDDL planning domain for automated deployment and reconfiguration of software in a distributed system. Satzger et al. [23] introduced a PDDL based planning approach for organic computing systems. Sykes et al. present in [13] an approach for planning architectural reconfiguration of component based systems based on the aforementioned three layer-architecture. They employ model checking as a planning technique based on labeled transition systems. This allows them to compute reactive plans, which generates actions sequence from every state in the state spaces towards the goal state.

None of the approaches supports the system developer appropriately in defining the required planning domains and offer techniques to check the correctness of the defined planning domain. A development process based on graph transformation naturally enables the support of respective modeling and verification tools and methods.

Vaquero et al. [27] present an approach for transforming UML models, use cases, class diagrams, state machines, and timing diagrams, to Petri Nets as well as PDDL in order to facilitate analysis and testing of requirements. The shown mapping from class diagrams to PDDL is similar to the one presented in this paper. In contrast to the approach by Vaquero et al., we use graph transformations for the specification of behavior which are more suitable for the specification of architectural reconfiguration.

There are only few tools and methodologies which support the designer in developing a planning model that complies to certain properties. For instance, Howey [15] et al. introduce VAL an automated tool that checks if plans generated by a planning system satisfy the specification made in the corresponding PDDL domain. Differently, PDVer is a tool that can be applied to check the correctness of Planning Domains [20]. However, PDVer does not formally verify the state transitions enabled by the planning domain, but heuristically generates and executes a number of test cases.

There are only a few approaches in the area of automated planning with graph transformations [6,8]. Edelkamp and Rensink present in [6] the combination of graph transformation and planning. They report that the employed planner (FF) can handle significantly bigger models than the graph transformation tool Groove. In contrast to our paper, Edelkamp and Rensink do not present how to automatically translate graph transformations to the input language of the employed planner. Estler [8] uses an A* as well as a Best First search for the computation of plans based on graph transformations. Instead of developing an own algorithm for planning, we employ standard off-the-shelf planning software which enables us to exploit their good performance and rich modeling properties, e.g., for temporal planning.

8 Conclusions and Future Work

We presented how graph transformation was used to specify actions for self-adaptive systems and how we use standard off-the-shelf automated planners to compute reconfiguration plans which order the execution of the reconfiguration actions. As a specific case of self-adaption we illustrated our approach by a self-healing process. We extended the employed story pattern formalism by several additional annotations for the specific case of durative actions in temporal planning. Based on this extensions, we showed how we translate story pattern to the Planning Domain Definition Language (PDDL) which is the standard planning language.

We have partially implemented the translation using the Eclipse Modeling Framework and Xpand as model-to-text translation environment. We used the EMF-based version of Fujaba, which is currently under development. We are currently working on finishing the implementation of the presented translation of attribute expressions as well as all syntax elements which are related to temporal planning.

Durative actions in the PDDL also include continuous effects which specify the continuous change of values during execution of the action, e.g., the physical position of an autonomous car based on its speed. It remains to be seen whether it makes sense to add those aspects to story patterns. Adding this might lead to a hybrid graph transformation formalism analogous to hybrid automata [1].

To reflect the specific strength and weaknesses of different planners as well as the differing requirements of application domains, it is reasonable to provide different translation schemes for story patterns to PDDL. The implementation and comparison of these different translations with respect to their effect on planners is an important part of our future research.

Story diagrams add control flow to story patterns. In order to use story diagrams for self-healing, we have to translate the control flow to PDDL as well. For the case of non-temporal planning, this works by numbering all story patterns and adding a sequence function which stores the current activity number. The control flow is then translated to appropriately handling this function in the precondition and the effect.

Acknowledgments. We thank Steffen Ziegert, Julian Suck, Florian Nafz, and Hella Seebach for discussions about the topic. We thank Christopher Gerking for the implementation of the prototypical translation of story patterns to PDDL as well as Alexander Stegmeier for the implementation of the simulation environment. Matthias Tichy was member of the software engineering group at the University of Paderborn, Germany, and the organic computing group at the University of Augsburg, Germany, while developing this approach. Benjamin Klöpper is a visiting researcher at NII and scholarship holder of the German Academic Exchange Service (DAAD).

References

1. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P.H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theoretical Computer Science 138(1), 3–34 (1995)
2. Arshad, N., Heimbigner, D., Wolf, A.: Deployment and dynamic reconfiguration planning for distributed software systems. Software Quality Journal 15(3), 265–281 (2007)
3. Becker, B., Beyer, D., Giese, H., Klein, F., Schilling, D.: Symbolic invariant verification for systems with dynamic structural adaptation. In: Proc. of the 28th International Conference on Software Engineering, pp. 72–81. ACM Press (2006)
4. Becker, B., Giese, H.: Modeling of correct self-adaptive systems: A graph transformation system based approach. In: Proc. of the 5th International Conference on Soft Computing as Transdisciplinary Science and Technology, pp. 508–516. ACM, New York (2008)
5. Chen, Y., Wah, B.W., Hsu, C.W.: Temporal planning using subgoal partitioning and resolution in SGPlan. J. Artif. Intell. Research 26, 323–369 (2006)
6. Edelkamp, S., Rensink, A.: Graph transformation and AI planning. In: Edelkamp, S., Frank, J. (eds.) Knowledge Engineering Competition. Australian National University, Canberra (2007)
7. Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.): TAGT 1998. LNCS, vol. 1764. Springer, Heidelberg (2000)
8. Estler, H.C., Wehrheim, H.: Heuristic search-based planning for graph transformation systems. In: Proc. of the Workshop on Knowledge Engineering for Planning and Scheduling (KEPS 2011), pp. 54–61 (2011)
9. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story diagrams: A new graph rewrite language based on the Unified Modeling Language and Java. In: Ehrig et al. [7], pp. 296–309
10. Fox, M., Long, D.: PDDL2.1: An extension to PDDL for expressing temporal planning domains. J. Artif. Intell. Research 20, 61–124 (2003)
11. Fürst, S., Mössinger, J., Bunzel, S., Weber, T., Kirschke-Biller, F., Heitkämper, P., Kinkelin, G., Nishikawa, K., Lange, K.: AUTOSAR – A worldwide standard is on the road. In: Proc. of the 14th International VDI Congress Electronic Systems for Vehicles 2009. VDI (2009)
12. Ghallab, M., Nau, D., Traverso, P.: Automated Planning – Theory and Practice. Morgan Kaufmann Publishers (2004)
13. Heaven, W., Sykes, D., Magee, J., Kramer, J.: A Case Study in Goal-Driven Architectural Adaptation. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Self-Adaptive Systems. LNCS, vol. 5525, pp. 109–127. Springer, Heidelberg (2009)
14. Heinzemann, C., Suck, J., Eckardt, T.: Reachability analysis on timed graph transformation systems. ECEASST 32 (2010)
15. Howey, R., Long, D., Fox, M.: VAL: Automatic plan validation, continuous effects and mixed initiative planning using PDDL. In: Proc. of the Int. Conference on Tools with Artificial Intelligence, pp. 294–301. IEEE Computer Society (2004)
16. Klöpper, B., Honiden, S., Meyer, J., Tichy, M.: Planning with utilities and state trajectories constraints for self-healing in automotive systems. In: Proc. of the 4th IEEE International Conference on Self-Adaptive and Self-Organizing Systems, pp. 74–83. IEEE Computer Society (2010)

17. Kramer, J., Magee, J.: Self-managed systems: An architectural challenge. In: Future of Software Engineering 2007, pp. 259–268. IEEE Computer Society (2007)
18. Le Métayer, D.: Describing software architecture styles using graph grammars. IEEE Transactions on Software Engineering 24(7), 521–533 (1998)
19. Oreizy, P., Medvidovic, N., Taylor, R.N.: Architecture-based runtime software evolution. In: Proc. of the 20th International Conference on Software Engineering, pp. 177–186. IEEE Computer Society (1998)
20. Raimondi, F., Pecheur, C., Brat, G.: PDVer, a tool to verify PDDL planning domains. In: Proc. of ICAPS 2009 Workshop on Verification and Validation of Planning and Scheduling Systems (2009)
21. Rensink, A.: The GROOVE Simulator: A Tool for State Space Generation. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 479–485. Springer, Heidelberg (2004)
22. Rozenberg, G.: Handbook of Graph Grammars and Computing by Grah Transformation, vol. 1: Foundations. World Scientific (1997)
23. Satzger, B., Pietzowski, A., Trumler, W., Ungerer, T.: Using Automated Planning for Trusted Self-organising Organic Computing Systems. In: Rong, C., Jaatun, M.G., Sandnes, F.E., Yang, L.T., Ma, J. (eds.) ATC 2008. LNCS, vol. 5060, pp. 60–72. Springer, Heidelberg (2008)
24. Stallmann, F.: A Model-Driven Approach to Multi-Agent System Design. Ph.D. thesis, University of Paderborn (2009)
25. Taentzer, G., Goedicke, M., Meyer, T.: Dynamic change management by distributed graph transformation: Towards configurable distributed systems. In: Ehrig et al. [7], pp. 179–193
26. Tichy, M., Henkler, S., Holtmann, J., Oberthür, S.: Component story diagrams: A transformation language for component structures in mechatronic systems. In: Postproc. of the 4th Workshop on Object-oriented Modeling of Embedded Real-Time Systems. HNI Verlagsschriftenreihe (2008)
27. Vaquero, T.S., Silva, J.R., Ferreira, M., Tonidandel, F., Beck, J.C.: From requirements and analysis to PDDL in itSIMPLE3.0. In: Proc. of the International Competition on Knowledge Engineering for Planning and Scheduling (2009)
28. Wermelinger, M., Fiadeiro, J.L.: A graph transformation approach to software architecture reconfiguration. Sci. Computer Programming 44(2), 133–155 (2002)
29. Zündorf, A.: Rigorous Object Oriented Software Development. University of Paderborn (2002)