

**Forschungsberichte  
der Fakultät IV – Elektrotechnik und Informatik**

**Modeling and Analysis  
of Self-Adaptive Systems  
Based on Graph Transformation**

Antonio Bucchiarone<sup>2</sup>  
Hartmut Ehrig<sup>1</sup>  
Claudia Ermel<sup>1</sup>  
Patrizio Pellicione<sup>3</sup>  
Olga Runge<sup>1</sup>

<sup>1</sup> Institut für Softwaretechnik und Theoretische Informatik  
Technische Universität Berlin, Germany  
{hartmut.ehrig, claudia.ermel, olga.runge}@tu-berlin.de

<sup>2</sup> Fondazione Bruno Kessler, Trento, Italy  
bucchiarone@fbk.eu

<sup>3</sup> Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica,  
Universita dell'Aquila, Italy.  
patrizio.pelliccione@gmail.com



# Modeling and Analysis of Self-Adaptive Systems Based on Graph Transformation

Antonio Bucchiarone<sup>2</sup>, Hartmut Ehrig<sup>1</sup>, Claudia Ermel<sup>1</sup>, Patrizio Pellicione<sup>3</sup>, Olga Runge<sup>1</sup>

<sup>1</sup> Institut für Softwaretechnik und Theoretische Informatik  
Technische Universität Berlin, Germany

hartmut.ehrig@tu-berlin.de, claudia.ermel@tu-berlin.de, olga.runge@tu-berlin.de

<sup>2</sup> Fondazione Bruno Kessler, Trento, Italy  
bucchiarone@fbk.eu

<sup>3</sup> Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica  
Università dell'Aquila, Italy.  
patrizio.pelliccione@gmail.com

**Abstract.** Software systems nowadays require continuous operation despite changes both in user needs and in their operational environments. Self-adaptive systems are typically instrumented with tools to autonomously perform adaptation to these changes while maintaining some desired properties. In this paper we model and analyze self-adaptive systems by means of typed, attributed graph grammars. The interplay of different grammars representing the application and the adaptation logic is realized by an adaption manager. Within this formal framework we define consistency and operational properties that are maintained despite adaptations and we give static conditions for their verification. The overall approach is supported by the AGG tool that offers the features for modeling, simulating, and analyzing graph transformation systems. A case study modeling a business process that adapts to changing environment conditions is used to demonstrate and validate the formal framework.

*Keywords:* self-adaptive systems, formal analysis, verification, graph transformation, AGG

## 1 Introduction

Continuous operation is required for modern complex software systems (e-commerce, avionics, logistic systems, etc.) despite changes both in user needs and in their operational environment. The high degree of variability that characterizes modern systems requires to

design them keeping runtime evolution in mind. Self-adaptive systems are systems that autonomously decide (e.g., without or with minimal interference) how to adapt the system at runtime to the internal reconfiguration and optimization requirements or to environment (context) changes and threats [10]. To achieve this goal, a self-adaptive system should be able to *monitor* itself and its context, to *detect* context changes that require system adaptations, to *decide* how to react and *act* to execute such decisions [44].

On their common basis of self-awareness, self-monitoring and context-awareness, self-adaptive systems are further classified by their characteristics, known as *self-\* properties* [31,2]. The initial four self-\* properties of self-adaptive systems are self-configuration, self-healing<sup>4</sup>, self-optimization, and self-protection [31]. Self-configuration comprises components installation and configuration based on some high-level policies. Self-healing deals with automatic discovery of system failures, and with techniques to recover from them. Typically, the runtime behavior of the system is monitored to determine whether a change is needed. The main objectives of self-healing are to maximize the availability, maintainability, survivability, and reliability of the system [23]. Self-optimization monitors the system status and adjusts parameters to increase performance when possible. Finally, self-protection aims to detect external threats and to mitigate their effects [48].

Even with good reactions to both system and context changes a set of high-level goals “should be maintained regardless of the environment conditions” [17]. In other words, the joint ability of effectively reacting to changes without degrading the level of dependability is a key factor for delivering successful systems that continuously satisfy evolving user requirements. Consequently, a reliable support for the consistent evolution of systems at runtime should be conceived through a well-defined formalization that provides a solid basis for analysis.

In [13], the authors modeled and verified dynamic software architectures and self-healing systems (called self-repairing systems in [13]), by means of hypergraphs and graph grammars. The work in [12] shows how to formally model self-healing systems by using algebraic graph transformations [19] and to prove consistency and operational properties. Graph transformation has been investigated as a fundamental concept for specification, concurrency, distribution, visual modeling, simulation and model transformation [19,20].

In this paper we extend the work in [12] by formally modeling and analyzing self-adaptive systems based on the framework of algebraic graph transformation. Since we aim at modeling in a general way the concepts of self-awareness, context-awareness, self-monitoring and self-adaptation, our modeling framework is in principle applicable to systems with different kinds of self-\* properties. The aim of our analysis is to show operational properties of self-adaptive systems concerning overall conflicts and dependencies of normal system behavior and adaptations. Hence, the analysis is not tailored to specific desired properties concerning e.g., security aspects in self-protective systems or performance analysis of self-optimization.

Self-adaptive systems are modeled in our approach as a set of typed graph grammars where three kinds of system rules are distinguished: normal, context, and adaptation rules.

---

<sup>4</sup> following [40] we consider self-healing and self-repair as synonyms.

Normal rules define the normal and ideal behavior of the system. States of self-adaptive systems fulfill consistency properties in the sense that all reachable states are system-consistent and all normal rules preserve and reflect normal states. Context rules define context flags (adaptation hooks) that trigger adaptation rules. Adaptation rules in different adaptation grammars define the adaptation logic.

The formalization enables the specification, analysis and verification of operational properties of self-adaptive systems. Operational properties define (i) when a system in an adaptation state can be adapted in a system-*enhancing* way to be in a normal state again, (ii) if the nature of the adaptation is *corrective*, i.e., the system state before adaptation can be recovered. Operational properties can be checked statically for the given system rules in an automatic way using the AGG<sup>5</sup> modeling and analysis tool for typed attributed graph transformation systems.

Summarizing, the contribution of this paper is twofold: (i) we propose a formal framework to model, simulate and analyze self-adaptive systems, where the adaptation is triggered by events monitored at the application layer and executed automatically; (ii) we provide a methodology to employ static analysis techniques supported by the tool AGG for analyzing self-adaptive system models. The theory is presented by use of a running example, a car logistics scenario in a seaport terminal.

The paper is organized as follows. Section 2 presents our running example. Section 3 introduces the framework for modeling and analyzing self-adaptive systems. Section 4 models the car logistics scenario by using the algebraic graph transformation framework. Section 5 describes how to formally verify desirable consistency and operational properties of self-adaptive systems. Section 6 discusses aspects related to the automation of the approach. Section 7 compares the approach proposed in this paper with related work. We conclude the paper in Section 8 with a summary and an outlook on future work.

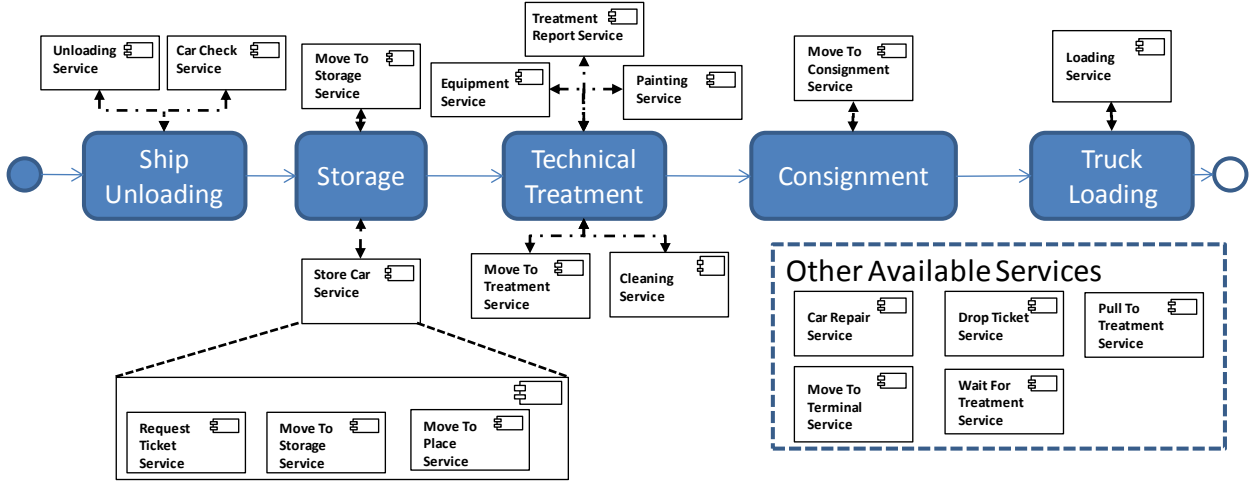
## 2 Running example

In this section we describe the *Car Logistics System (CLS)* scenario that will be used throughout the paper to explain the approach. At the automobile terminal of the Bremerhaven seaport [8], nearly 2 million of new vehicles are handled each year; the business goal is *to deliver them from the manufacturer to the dealer*. To achieve that, several intermediate business activities are involved. These include unload and store cars from a ship, apply to them treatments to meet the customer’s requirements and distribute them to the retailers. The company “Logistics IT Solutions” wants to develop a service-based application (the CLS) to support the delivery of vehicles from the ship to the retailers. The CLS must implement the business process depicted in Figure 1 by invoking and orchestrating the set of available services in a proper way. Each business activity of the process is executed invoking a set of available services (i.e., Car Check Service, Unloading Service, etc.) that can be atomic or composite

---

<sup>5</sup> AGG (Attributed Graph Grammars): <http://www.tfs.tu-berlin.de/agg>.

(i.e., Store Car Service). Additional services, i.e., services that are not directly attached to the business process, are defined and they can be used during the application execution. For example, the Wait For Treatment Service may be invoked when a vehicle that needs a treatment has to wait some time because of a long queue in the treatment station.



**Fig. 1.** Business Process and Services of the Car Logistics Scenario.

The CLS executes the business process presented before for each vehicle under the following assumptions: (i) each business activity is executed in the defined order; (ii) the context in which the business process is executed can evolve in time.

Assume now the following two cases that can happen at run-time:

- *A vehicle is severely damaged during its movement from the ship to the storage area:* The vehicle has been unloaded from the ship and has requested a ticket (using the Request Ticket Service) to park in the storage area. It receives a precise ticket and starts to move to the storage (using the Move To Storage Service). While moving, the vehicle is severely damaged and then it stops. In this case the business process does not know how to proceed and the booked ticket cannot be used.
- *A vehicle arrives at a service point in the treatment area, but the required service is busy and cannot be executed immediately:* The vehicle is ready to undergo a service as ordered by the customer (concerning e.g. painting or equipment), but there are already a number of vehicles waiting for this service. In this case, the corresponding business service cannot proceed in an expected way (such as treating the vehicles in order of their arrival).

As shown in the previous cases, there are situations in which the business process cannot proceed according to the defined CLS execution. The main reasons for that are: (i) some specific business process variants have not been specified at design time (e.g., due to some

error) and (ii) it is not possible to predict a priori which variants should be followed (due to lack of information on the execution contexts). In order to be able to execute the CLS also in case of unexpected situations we need to address the following problems:

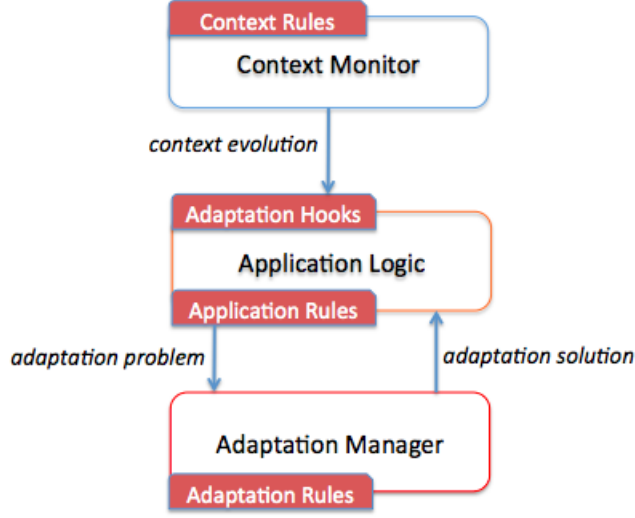
- *Context-awareness*: In many applications the role of the context is fundamental in realizing the adaptation functionalities [1]. In order to relate the application execution to the context the service-based application must be context-aware. This means that during the execution it must be possible to obtain information on the underlying environment (e.g., relevant information on world entities involved, status of the business process execution, human activities, etc.). To be adaptable, an application should provide adaptation hooks, i.e., information to the environment on part of its structure and its behavior. This context information provided by the adaptation hooks should be used to select the most suitable adaptation strategy.
- *Separation of Concerns*: The adaptation logic should be developed separately from the application logic by some adaptation engineer, for instance as a set of adaptation rules. The adaptation logic can be created and/or changed after the application has been deployed without modifying the running application. This coordination work is done by an adaptation manager that requires human intervention for selecting the most suitable adaptation and for adding new adaption rules. At runtime, the adaptation manager should check both context and user needs, control whether any adaptations have to be applied to the application, and exploit the adaptation hooks provided by the application to reconfigure it in the desired way.

### 3 Framework for Rule-Based Dynamic Adaptation

The framework manages the dynamic adaptation by means of rules specifying when and how adaptation is triggered, how the choice among the possible adaptations is performed, and finally how the nature of adaptations can be characterized.

**Components of the adaptation framework** Our adaptation framework is composed of three fundamental components as shown by Figure 2: *Context Monitor* that describes properties on the application operational environment and how they evolve (i.e., *context rules*), *Application Logic* that describes how the application evolves (i.e., *application rules*). Finally, *Adaptation Manager* that specifies how a system is adapted in case of adaptation needs (i.e., *adaptation rules*).

According to the scenario presented in section 2, the considered self-adaptive system is the Car Logistics application. Its application logic describes what are the different activities that can be executed (i.e., Ship Unloading, Storage, Technical Treatment, Consignment, and Truck Loading), the set of available services that can be used to realize such activities (i.e., Store Car Service, Move To Treatment Service, Cleaning Service, etc.) and the assumed behavior of the overall application. The behavior describes the precise order of the activities that a car must execute plus a precise set of business policies (in terms of activity preconditions) to respect.



**Fig. 2.** Components of the Adaptation Framework.

The *Context Monitor* continuously monitors the context at fixed intervals. The context monitor is defined as a set of small rules (called *context rules*) which once applied, add so-called *adaptation hooks* to the system to trigger the adaptation process. The system is monitored at regular time intervals, and an *adaptation problem* is sent to the *Adaptation Manager* if one or more adaptation hooks are found. In response, the *Adaptation Manager* returns an *adaptation solution* to the application logic that aims to do its best to “recover”, so that the blocked activity can be executed and the main process can continue.

**Formalization of self-adaptive system behavior in the framework** In our formalization, the formal model of a self-adaptive system is a set of graph grammars typed over the same type graph. A main system grammar consisting of *system rules* modeling normal behavior (the *Application Logic*) and *context rules* modeling changes that require adaptation by generating adaptation hooks, can be enhanced by other grammars, each representing a specific adaptation. Such an adaptation grammar contains *adaptation rules* modeling reactions to the detection of context changes. *Context Monitoring* is modeled by context constraints in the main system grammar that are violated in the presence of adaptation hooks and trigger the (semi-automatic) selection of a corresponding adaptation grammars (the *Adaptation Logic*). The interplay of the different grammars representing the adaptation logic is realized by the *Adaptation Manager*.

The adaptation is triggered on activity enter, i.e., whenever a vehicle enters a new service, the vehicle and the service are monitored for change requirements.

*Remark 1 (Context-aware and self-aware systems).* Note that we do not distinguish between *context-aware* and *self-aware* systems. In our approach, adaptation hooks are all issued



from context rules. It would be no problem in our approach to differentiate between two types of adaptation hooks with respect to their origin (context or the system itself). For this, we would need context rules and certain distinguished self-rules. But since we abstract from the context and the actual origin of the adaptation hook, the effect would remain the same, i.e., the adaptation hooks would lead to an adaptation of the system by applying an adequate adaptation grammar. Since, technically, this is not a limitation, we prefer to keep the modeling approach simple and to remain with context rules only.

**Ordering adaptations** Different adaptations may be applicable during the system execution. The choice of which adaptation to apply may influence the time required for performing the adaptation, or even the final result. In our framework, we prefer *controlled* application of adaptations, i.e. adaptations are selected by the adaptation manager or they may have been associated with a priority; if many adaptations can be applied, the one with highest priority may be applied first.

**Nature of adaptations** In order to be able to recover our applications at run-time, we consider two classes of adaptations that can be applied and treated in different ways [16,34], in particular:

- *Corrective Adaptation*: these adaptations take care of adapting the application when the current implementation instance cannot proceed with the execution in the current context (i.e., a car is damaged). The main objective of these adaptations is to recover the application and hence focuses on the *self-healing* property. This is achieved through an adaptation that, starting from the actual context state, performs the necessary changes in the domain to bring the application and its context to the expected state where it can be executed again. In our framework, an adaptation is *corrective* if each adaptation state can be adapted in a corrective way (repaired), i.e., the normal state before the adaptation became necessary is reestablished.
- *Enhancing Adaptation*: these adaptations enhance existing services of the application; this may for instance change the non-functional properties of the service, or provide new services with the same or enhanced functionalities. In our framework, an adaptation is *enhancing* if each adaptation state can be adapted to become a normal state, possibly by adding new functionalities and services. The normal state after the adaptation is *not necessarily* identic with the normal state before the adaptation became necessary.

## 4 Modeling self-adaptive systems by using Algebraic Graph Transformation and AGG

In this section, we show how to model self-adaptive systems in the formal framework of algebraic graph transformation (AGT) [19]. Specifically, typed graphs, introduced in Def. 1, are used to model the static part of the system. Typed graphs are enriched with constraints

that self-adaptive systems have to satisfy even during adaptation. Moreover, we model the behavior and the adaptation of self-adaptive systems by means of graph grammars, introduced in Def. 2. This section makes use of the *Car Logistics System* running example and of the AGG tool to show how practitioners can model self-adaptive systems.

**Definition 1 (Typed Graphs).** A graph  $G = (N, E, s, t)$  consists of a set of nodes  $N$ , a set of edges  $E$  and functions  $s, t : E \rightarrow N$  assigning to each edge  $e \in E$  the source  $s(e) \in N$  and target  $t(e) \in N$ .

A graph morphism  $f : G \rightarrow G'$  is given by a pair of functions  $f = (f_N : N \rightarrow N', f_E : E \rightarrow E')$  which is compatible with source and target functions, i.e.,  $f_N \circ s = s' \circ f_E$ , and  $f_N \circ t = t' \circ f_E$ .

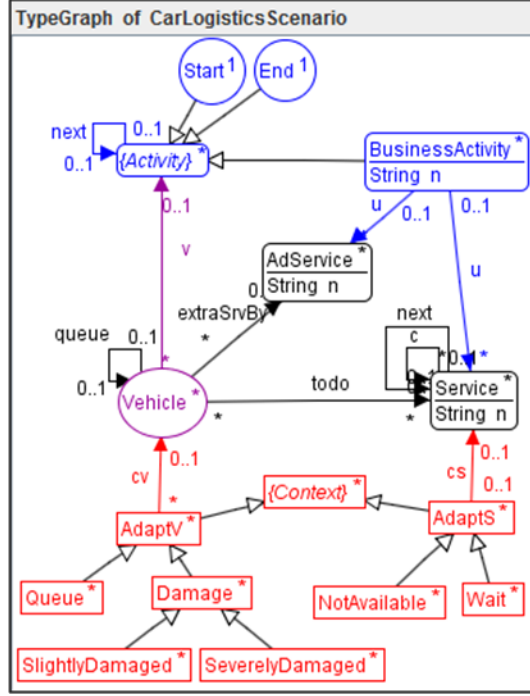
A type graph  $TG$  is a graph where nodes and edges are considered as node and edge types, respectively. A  $TG$ -typed, or short typed graph  $\overline{G} = (G, t)$  consists of a graph  $G$  and a graph morphism  $t : G \rightarrow TG$ , called typing morphism of  $G$ . Morphisms  $f : \overline{G} \rightarrow \overline{G'}$  of typed graphs are graph morphisms  $f : G \rightarrow G'$  which are compatible with the typing morphisms  $t : G \rightarrow TG$  and  $t' : G' \rightarrow TG$ , i.e.,  $t' \circ f = t$ .

For simplicity, we abbreviate  $\overline{G} = (G, t)$  by  $G$  in the following. Moreover, the approach is also valid for *attributed* and *typed attributed graphs* where nodes and edges can have data type attributes [19], as used in our running example.

*Example 1 (Type Graph and Initial State for Car Logistics).* Figure 3 shows the type graph for the Car Logistics case study. Note that we have an integrated type graph which contains types used for modeling the “normal” aspects of the car logistics scenario, as well as the context types used for adaptation, e.g., the hooks (context flags) that trigger the adaptation rules.

In the integrated type graph, we have the following types for normal behavior:

- **Start**, **End** and **BusinessActivity** are the main business activities (the colored nodes in Figure 1), which are ordered (linked by directed arcs of type `next`).
- **Service** is a service station belonging (linked) to a **BusinessActivity**. A service may be a composite service. Then it contains other services which are ordered (linked by `next` arcs). Containment of sub-services in a composite service is modeled by `c` edges from the sub-services to its composite service. To keep things simple, we currently allow only one nesting level (a sub-service cannot be a composite service).
- **Vehicle** is a car running through the business process. At the beginning it will be linked (by a `v` link) to the **Start** activity and is ready to enter a service.
- A `todo` link between a vehicle and each service of each **BusinessActivity** is generated when a **Vehicle** starts the business process. The successful processing of a service leads to the deletion of the corresponding `todo` link. When all services belonging to the business process have been processed (all `todo` links are removed), the **Vehicle** arrives at the **End** activity as a completed product with a precise treatment executed and ready to be delivered to a retailer.



**Fig. 3.** Type Graph of the Car Logistics Case Study

For adaptation handling we have the following types:

- **Context** is the super-type for all possible context signals, including adaptation hooks. These hooks are used for triggering the adaptation grammars. We specify two main context types **AdaptV** and **AdaptS**.
- **AdaptV** with refinements **Damage** and **Queue** denoting that a car is damaged and needs to be repaired (**SlightlyDamaged**) or disposed (**SeverelyDamaged**), or a car is in a queue.
- **AdaptS** with refinements **NotAvailable** and **Wait** denoting that a service station is not available (our case study does not cover this case) or there is a queue at a service, respectively. The refinement **Wait** denotes that cars in the current business activity should queue up and wait to be processed.
- **AdService** is an adaptation service not directly attached to a **BusinessActivity**. Adaptation services are additional services used during the business process execution according to an adaptation scenario.
- An edge of type **extraSrvBy** (extra service by) connects a **Vehicle** to an adaptation service.
- Edges of type **queue** connect the **Vehicles** in a queue at a service.

The graph modeling the initial state of the scenario with two vehicles waiting to be processed is shown in Figure 4.

In order to model consistency and adaptation constraints of a self-adaptive system, we use (*TG*-typed) graph constraints. A *graph constraint* is given by an injective graph morphism

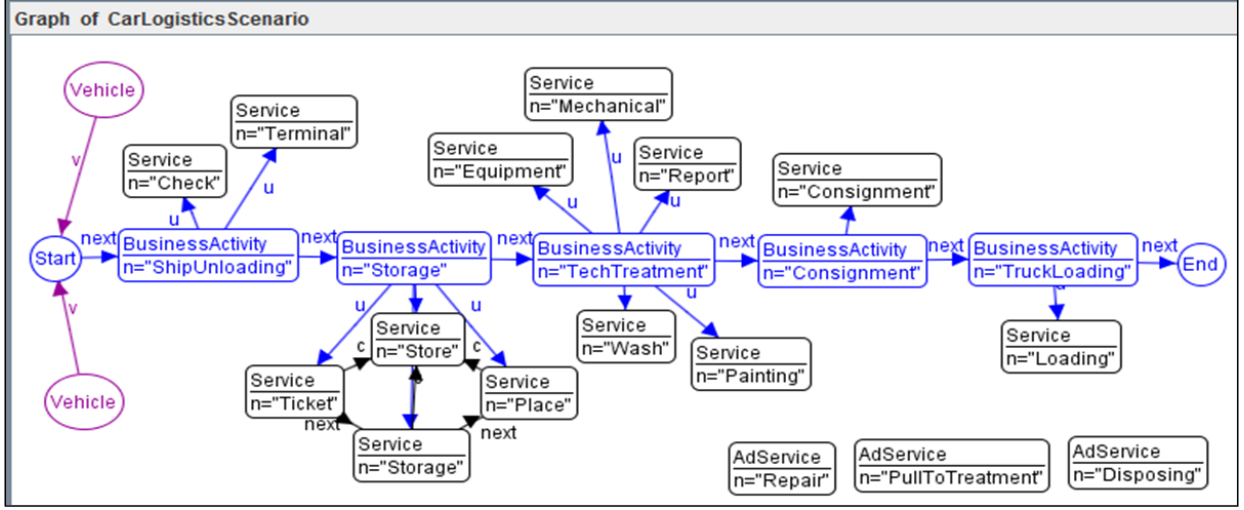


Fig. 4. Initial State Graph of the Car Logistics Case Study

$c : P \rightarrow C$  (where  $P$  is called *premise* and  $C$  *conclusion*). Constraint  $c : P \rightarrow C$  is *satisfied* by a graph  $G$ , written  $G \models c$ , if the existence of an injective graph morphism  $p : P \rightarrow G$  implies the existence of an injective graph morphism  $q : C \rightarrow G$ , such that  $q \circ c = p$ . Graph constraints can be negated or combined by logical connectors (e.g.,  $\neg c$ ). If  $P$  is empty, only the existence of  $C$  is required.

*Example 2 (Graph constraints for the Car Logistics system).* Here we define some system constraints that have to be satisfied throughout the states of the scenario:

$$\begin{aligned} C_{consist} &= \{noFalseServiceConnect, sameBAforComp, noEqualContextFlags\}, \\ C_{adapt}^1 &= \{Damage\}, \\ C_{adapt}^2 &= \{Wait\}. \end{aligned}$$

- $sameBAforComp = sameActivityOfCont \wedge sameActivityOfNext \wedge sameComposite$ : All subservices belonging to the same composite service are linked to the same BusinessActivity and to the same composite service node (Figure 5).
- $noFalseServiceConnect = \neg(next-loop \vee c-c-loop \vee c-next \vee todo-c)$ : There are no *next* or containment loops, and a subservice cannot be a composite service, and a vehicle cannot be served (*todo* edge) by a service which is a container of other services (Figure 6).

Moreover, we have adaptation constraints, describing adaptation hooks that are required to hold for certain adaptations to occur. Adaptation constraint *Wait* requires the existence of a *Wait* flag at a service, and adaptation constraint *Damage* requires a *Damage* flag at a vehicle (Figure 7).

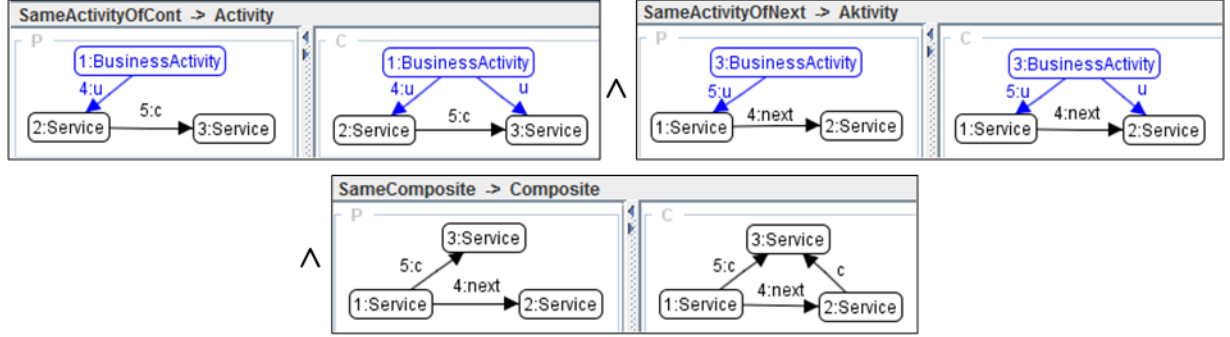


Fig. 5. Graph constraint *sameBAforComp*.

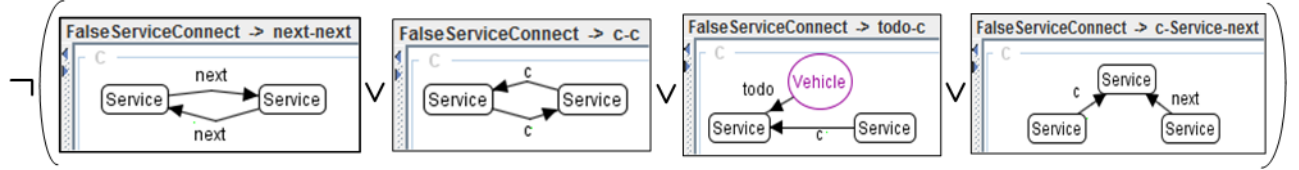


Fig. 6. Graph constraint *noFalseServiceConnect*.

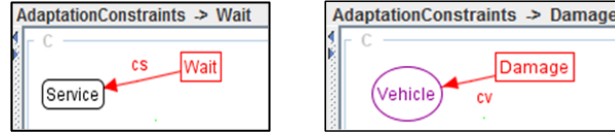


Fig. 7. Adaptation constraints *Wait* and *Damage*.

We model the behavior of the main scenario and the adaptations in different graph grammars. Whenever an adaptation becomes necessary, the respective adaptation rules are loaded into the main case study grammar.

**Definition 2 (Typed Graph Grammar).** A typed graph grammar  $GG = (TG, G_{init}, Rules)$  consists of a type graph  $TG$ , a  $TG$ -typed initial graph  $G_{init}$  and a set of graph transformation rules ( $Rules$ ). Each rule  $r \in Rules$  is given by a span  $(LHS \leftarrow I \rightarrow RHS)$ , where  $LHS$ ,  $I$  and  $RHS$  are  $TG$ -typed graphs, called left-hand side, interface and right-hand side, respectively. Moreover,  $I \rightarrow LHS$ ,  $I \rightarrow RHS$  are injective typed graph morphisms where in most cases  $I$  can be considered as the intersection of  $LHS$  and  $RHS$ . A rule  $r \in Rules$  is applied to a  $TG$ -typed graph  $G$  by a match morphism  $m : LHS \rightarrow G$  leading to a direct transformation  $G \xrightarrow{r,m} H$  via  $(r, m)$  in two steps: first, we delete the match  $m(LHS)$  without  $m(I)$  from  $G$  to obtain a context graph  $D$ , and then, we glue together  $D$  with  $RHS$  along  $I$ , i.e., we construct a union of  $D$  and  $RHS$  with the intersection graph  $I$ , leading to a  $TG$ -typed graph  $H$ . This gluing construction is represented in the diagram below<sup>6</sup>, where diagram (1) (resp.

<sup>6</sup> Formally, the squares (1) and (2) are called *pushouts* in the category  $\mathbf{Graphs}_{TG}$  of  $TG$ -typed graphs.

(2)) corresponds to gluing  $G$  of  $LHS$  and  $D$  along  $I$  (resp. to gluing  $H$  of  $RHS$  and  $D$  along  $I$ ).

$$\begin{array}{ccccc}
 N & \xleftarrow{nac} & LHS & \xleftarrow{l} & I & \xrightarrow{r} & RHS \\
 & \searrow q & \downarrow m & & \downarrow (1) & & \downarrow (2) & \downarrow m^* \\
 & & G & \xleftarrow{\quad} & D & \xrightarrow{\quad} & H
 \end{array}$$

Note that diagram (1) in step 1 only exists if the match  $m$  satisfies a gluing condition with respect to rule  $r$  which makes sure that the deletion in step 1 leads to a well-defined  $TG$ -typed graph  $D$ , leaving no dangling edges. Moreover, rules are allowed to have one or more Negative Application Conditions (NACs). A NAC is a negated graph constraint  $nac : L \rightarrow N$ . A rule  $r$  with a NAC can only be applied at match  $m : L \rightarrow G$  if this match satisfies the negated constraint, i.e., there is no injective morphism  $q : N \rightarrow G$  with  $q \circ nac = m$ . This means intuitively that  $r$  cannot be applied to  $G$  if graph  $N$  occurs in  $G$ . A transformation  $G_0 \xRightarrow{*} G_n$  via Rules in  $GG$  consists of  $n \geq 0$  direct transformations  $G_0 \Rightarrow G_1 \Rightarrow \dots \Rightarrow G_n$  via rules  $r \in Rules$ . For  $n \geq 1$  we write  $G_0 \xRightarrow{+} G_n$ , for  $n \geq 0$  we write  $G_0 \xRightarrow{*} G_n$ .

The normal behavior and adaptations of our case study are modeled by typed graph grammars as follows. In the main grammar *CarLogisticsScenario* (Example 3), normal behavior is modeled. Moreover, context flags (e.g., adaptation hooks) can be generated. The adaptation hooks trigger the adaptation logic, modeled by adaptation grammars, e.g., *Repair-Adaptation* (Example 5) and *Wait-At-Queue-Adaptation* (Example 7). We may have different adaptation grammars that are suitable for the same adaptation hook. For instance, if a damaged car is in the midst of a composite service, first a rollback adaptation has to be performed, and then a repair adaptation. The adaptation manager coordinates the different adaptation grammars such that the rules of the most suitable adaptation grammars are imported into the main grammar. These rules perform the necessary adaptations at the host graph so that the “normal behavior rules” can proceed and maybe new adaptation hooks are set. Note that in AGG the modeler plays the role of an adaptation manager and imports the necessary adaptation rules into the main grammar. However, the verification is incremental, i.e., verifications previously performed are taken into account and only newly added rules are required to be checked for conflicts with respect to the existing ones. After adaptation, the adaptation rules are removed from the main grammar, and the application of the “normal behavior rules” continues.

*Example 3 (Rules modeling the Car Logistics scenario).* Here, we model the rules for the normal behavior of Vehicles running through BusinessActivities smoothly. To save space, we depict only the left- and right-hand sides for each rule. The interface consists of those nodes and edges that are present both in  $LHS$  and in  $RHS$  and mapped to each other by equal numbers. For each rule, a unique rule name is depicted on top of the  $LHS$ , and the NACs are placed around the  $LHS$ . The normal behavior rules are shown in Figure 8 to 13.

The first step for each Vehicle is to enter the business process. Then, all services of the Vehicle’s BusinessActivities are marked as *to do* by creating `todo` edges between the Vehicle

and each service not yet marked (applying rule **ServiceToDo** in Figure 8 as long as possible). Now rule **EnterBP** in Figure 9 moves the Vehicle to the first BusinessActivity. When a service that is not composite is processed, the corresponding **todo** edge is removed by rule **DoService** in Figure 10.

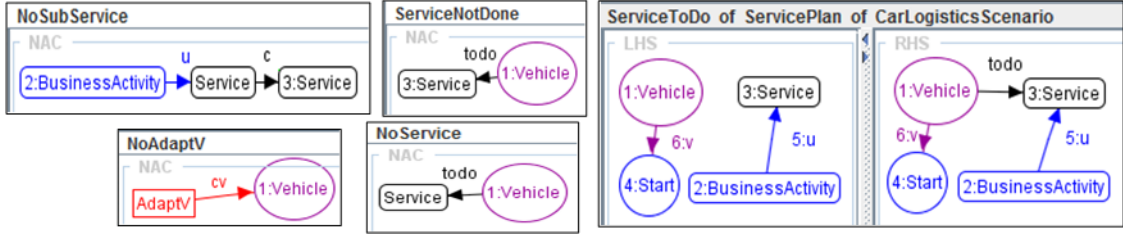


Fig. 8. The normal behavior rule **ServiceToDo**.

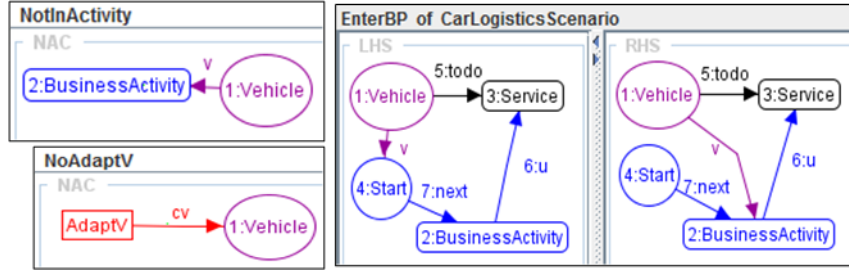


Fig. 9. The normal behavior rule **EnterBP**.

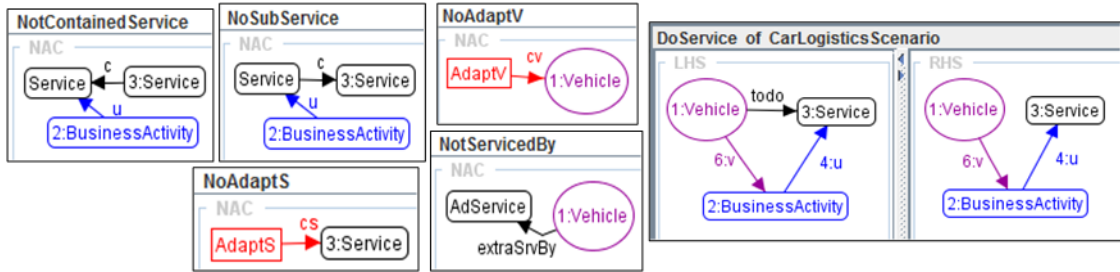


Fig. 10. The normal behavior rule **DoService**.

For processing a composite service consisting of ordered sub-services, the NAC of rule **DoSubService** in Figure 11 ensures that a sub-service is processed only if its previous services in the queue have already been processed.

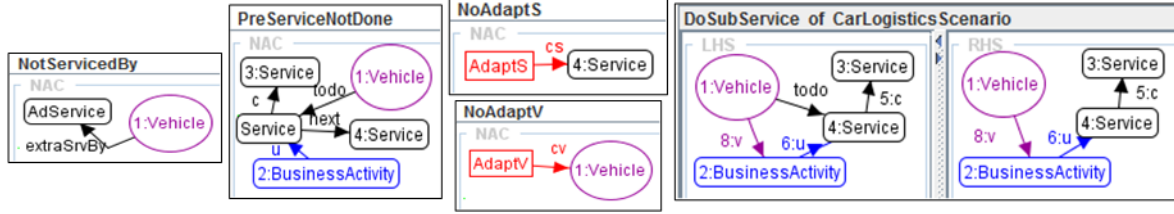


Fig. 11. The normal behavior rule DoSubService.

A Vehicle can move to the next BusinessActivity when all services of the previous one are done (rule NextBA in Figure 12). Finally, when there are no more services to do, the business process for the Vehicle is finished (rule FinishBP in Figure 13).

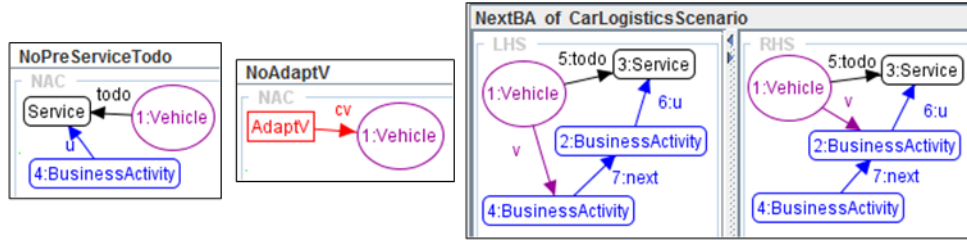


Fig. 12. The normal behavior rule NextBA.

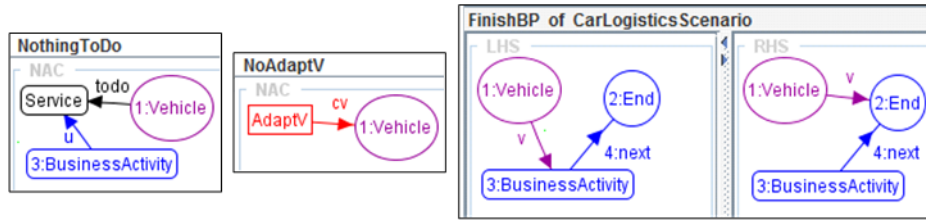
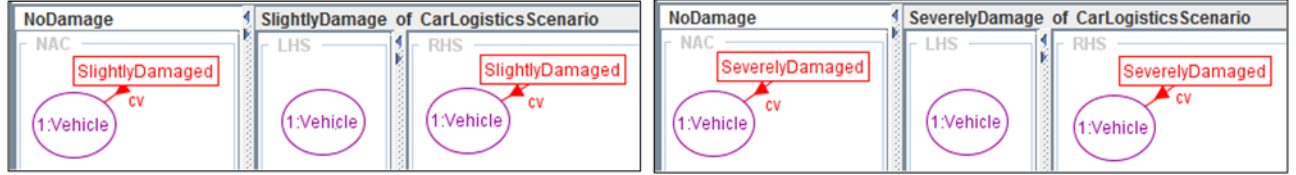


Fig. 13. The normal behavior rule FinishBP.

In addition to the *normal behavior* rules, the main grammar contains *context rules* that are applicable at any time and mark a Vehicle or Service with an adaptation hook, i.e., they create a node of one of the context node types AdaptV or AdaptS, respectively. In case a car is damaged, rules SlightlyDamage or SeverelyDamage mark it by an adaptation hook either of kind “SlightlyDamaged” or “SeverelyDamaged” as shown in Figure 14. Note that it is intended in our model, that a car that is slightly damaged can become severely damaged later (or the other way round, e.g. by being part of a crash while waiting to be repaired or disposed of). Of course, in this case it would make no sense that the slight damage of such

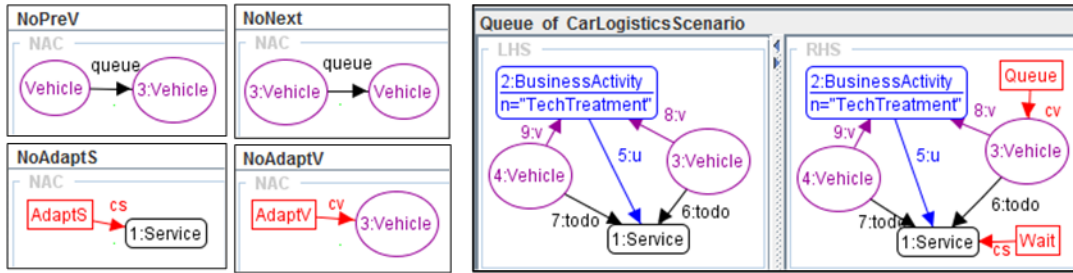


a car would be repaired before it would be disposed of. This is an example for a situation where the adaptation manager has to make the right choice between two possible adaptation grammars, the repair-adaptation (Example 5) or the dispose-adaptation (Example 6).



**Fig. 14.** The context rules *SlightlyDamage* and *SeverelyDamage*.

The context rule *Queue* in Figure 15 marks a service of the technical treatment area and a vehicle with context nodes of type *Wait* and *Queue*, respectively, if a service station in the technical treatment area is busy and the treatment cannot be executed immediately.



**Fig. 15.** The context rule *Queue*.

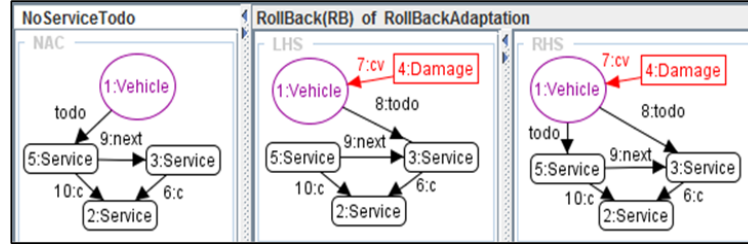
All above introduced adaptation hooks guide the adaptation manager to select a suitable adaptation grammar realizing the adaptation.

#### *Example 4 (Rollback-Adaptation).*

A rollback-adaptation becomes necessary when a damaged vehicle is in the midst of a composite service as shown in Figure 16. In this case, the already finished sub-services are “rolled back” by applying rule *RollBack* as long as possible before the vehicle is moved to the treatment area to be repaired.

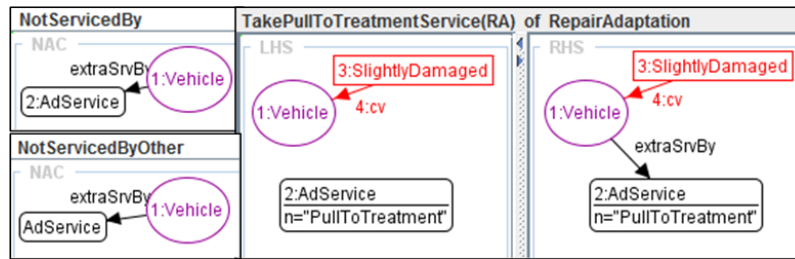
#### *Example 5 (Repair-Adaptation).*

A repair-adaptation becomes necessary when a vehicle is marked by a context flag as being slightly damaged. We require that the vehicle to be repaired is not in the midst of a composite service any more. (If it is, the rollback adaptation has to be selected by the adaptation manager to be performed before the repair adaptation.)



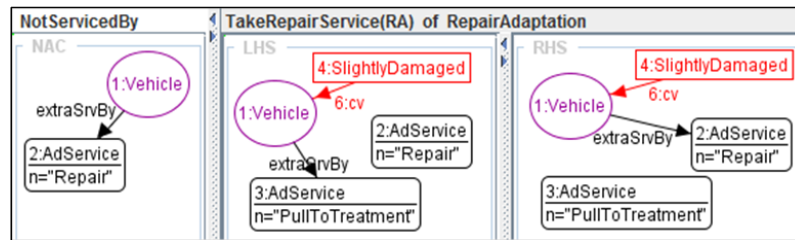
**Fig. 16.** The rollback-adaptation rule RollBack.

When reparation is needed, two additional services (which are not normal services of the usual business process) are evoked, i.e., the Vehicle is linked to them, one after the other. Rule **TakePullToTreatmentService** in Figure 17 uses an extra service to pull up the damaged Vehicle to the treatment area.



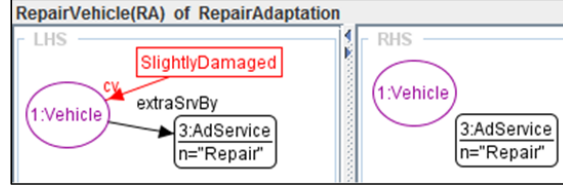
**Fig. 17.** The repair-adaptation rule TakePullToTreatmentService.

Rule **TakeRepairService** in Figure 18 allocates an extra repair service for the slightly damaged Vehicle.



**Fig. 18.** The repair-adaptation rule TakeRepairService.

A slightly damaged Vehicle is repaired (i.e., the adaptation hook is removed) by rule **RepairVehicle** in Figure 19.

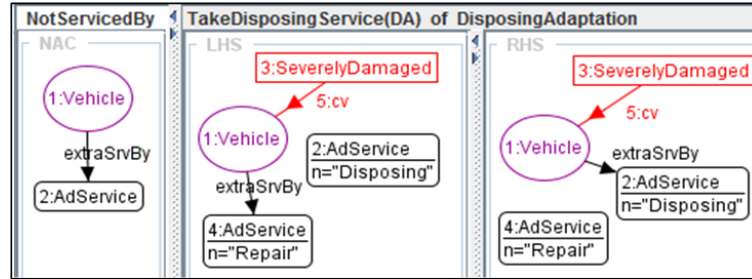


**Fig. 19.** The repair-adaptation rule *RepairVehicle*.

After repair, the Vehicle should continue its normal behavior at the point where the adaptation hook was set. The composite service it left before (and which has rolled back by applying the rollback-adaptation) may now start again from the beginning. By the way, the Vehicle does not forget which services have been done already and which are still to do.

*Example 6 (Dispose-Adaptation).*

A severely damaged vehicle that cannot be repaired, is disposed by the dispose-adaptation. This adaptation grammar also contains rules *TakePullToTreatmentService* and *TakeRepairService* that are analogous to the corresponding rules (Figure 17 and 18) in the repair-adaptation in Example 5 with the slight difference that the context flag is now always of kind *SeverelyDamaged*. Using rule *TakeDisposingService* in Figure 20, a severely damaged Vehicle is picked up by the disposing service.



**Fig. 20.** The dispose-adaptation rule *TakeDisposingService*.

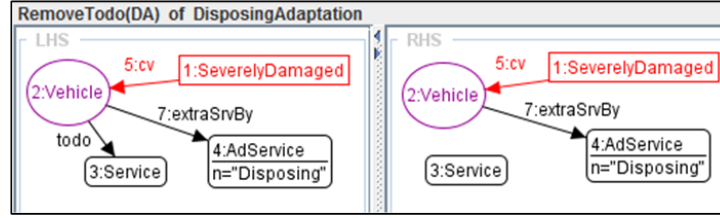
Before the vehicle can be disposed of, its *todo* links are removed by applying rule *RemoveToDo* in Figure 21 as long as possible.

Applying rule *RemoveDamageFlag* in Figure 22 as long as possible, all flags of kind *SlightlyDamaged* or *SeverelyDamaged* are removed from the vehicle.

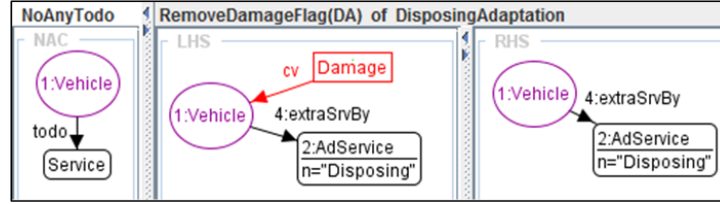
Finally, the vehicle is disposed of by rule *DisposeVehicle* in Figure 23 (in our model, a disposed vehicle remains in the graph without any links to other objects).

*Example 7 (Wait-At-Queue Adaptation).*

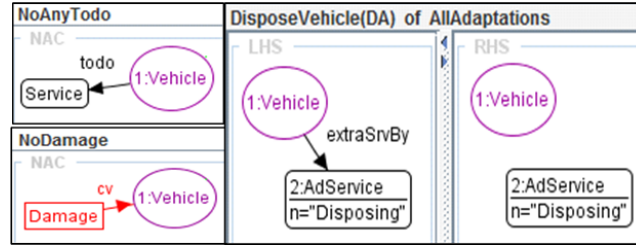
The wait-at-queue adaptation becomes necessary when a service station in the technical treatment area is busy and the treatment cannot be executed immediately. When a car



**Fig. 21.** The dispose-adaptation rule RemoveToDo.

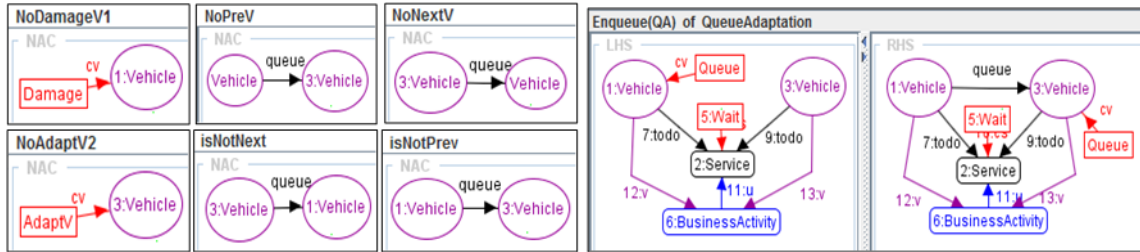


**Fig. 22.** The dispose-adaptation rule RemoveDamageFlag.



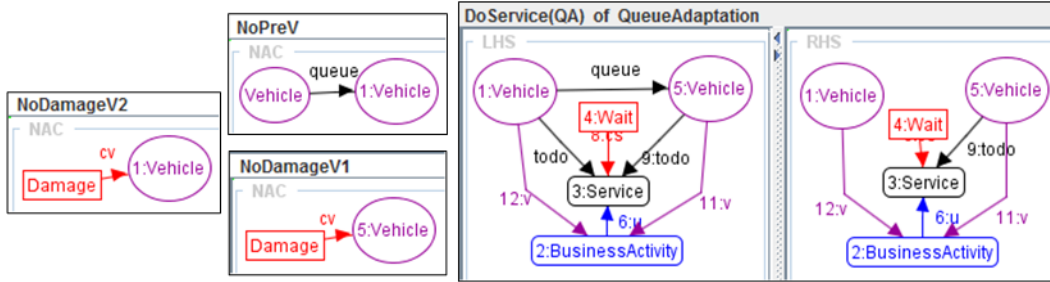
**Fig. 23.** The dispose-adaptation rule DisposeVehicle.

arrives at the treatment area and discovers that there is a queue it should queue up and wait. Rule **Enqueue** in Figure 24 enqueues all Vehicles waiting for this service. In doing so the Queue flag is shifted along the queue link.



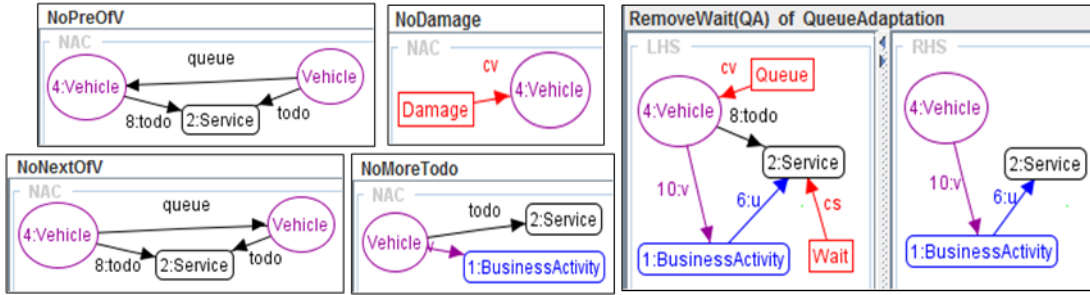
**Fig. 24.** The wait-at-queue-adaptation rule Enqueue.

Vehicles in a queue are served in the order of their arrival by applying rule **DoService** in Figure 25.



**Fig. 25.** The wait-at-queue-adaptation rule **DoService**.

Finally, the **Wait** flag at the busy service and the **Queue** flag at the last **Vehicle** are removed by rule **RemoveWait** in Figure 26 after the queue has been processed completely.



**Fig. 26.** The wait-at-queue-adaptation rule **RemoveWait**.

Thereafter, the normal behavior of vehicles running through **BusinessActivities** may proceed.

## 5 Formal Analysis of Self-Adaptive Systems

In this section we define self-adaptive systems in the framework of algebraic graph transformation (AGT) (subsection 5.1) and distinguish desirable operational properties of self-adaptive systems (subsection 5.2). In subsection 5.3, we analyze the operational properties introduced in subsection 5.2 and give static sufficient conditions for their verification.

### 5.1 Classification of Self-Adaptive System States

An adaptive system is defined in Definition 3 by a typed graph grammar where system rules can be partitioned into *normal*, *context* and *adaptation* rules. Moreover, we have two kinds of

$TG$ -typed graph constraints, namely *consistency* and *adaptation* constraints. Requirement 1 of Definition 3 ensures that context rules can be applied independently of the normal system behavior occurring in different parts of the system, which is expressed by the notion of sequential independence. Whenever the application of a normal rule is possible after the application of a context rule, the order of rule applications may be swapped, but the result after applying both rules remains the same. This requirement is important since we model systems that are monitored at regular time intervals. Hence, if a context rule has been applied first, and normal rules could be applied afterwards, they are applicable only to those objects of the system that are currently not in need of urgent adaptation (modeled by suitable NACs forbidding the presence of certain adaptation hooks). This models the situation in real-life systems that, e.g., damaged vehicles will not proceed in the normal business process (e.g., become painted) before they have been repaired. Even if the context change requires an adaptation that involves the shutdown and reboot of the system, it is more realistic to assume that some (restricted) normal behavior is possible after context changes have occurred and before the need for adaptation is discovered by the context monitor. In Requirement 2, we state that all reachable system states are consistent with respect to our consistency constraints. Requirement 3 concerns the interplay of normal behavior and adaptation behavior: normal rules must not create or insert adaptation hooks (i.e., only context rules may generate adaptation states) to trigger adaptations. Requirement 4 ensures that adaptation results are unique, i.e., the resulting state after applying adaptation rules does not depend on the order or location of their application.

**Definition 3 (Self-Adaptive System in AGT-Framework).**

A Self-adaptive system (SA-system) is given by  $SAS = (GG, C_{sys})$ , where:

- $GG = (TG, G_{init}, R_{sys})$  is a typed graph grammar with type graph  $TG$ , a  $TG$ -typed initial graph  $G_{init}$ , a set of  $TG$ -typed rules  $R_{sys}$  with NACs, called system rules, defined by  $R_{sys} = R_{norm} \cup R_{cont} \cup R_{adapt}$ , where  $R_{norm}$  (called normal rules),  $R_{cont}$  (called context rules) and  $R_{adapt}$  (called adaptation rules) are pairwise disjoint.
- $C_{sys}$  is a set of  $TG$ -typed graph constraints, called system constraints, with  $C_{sys} = C_{consist} \cup C_{adapt}$ , where  $C_{consist}$  (called consistency constraints) and  $C_{adapt}$  (called adaptation constraints) are pairwise disjoint.

We distinguish *reachable*, *consistent*, *adaptation* and *normal* states, where *reachable* states are partitioned into *normal* and *adaptation* states:

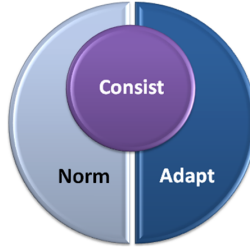
- $Reach(SAS) = \{G \mid G_{init} \xRightarrow{*} G \text{ via } R_{sys}\}$ , the reachable states consisting of all states reachable via system rules,
- $Consist(SAS) = \{G \mid G \in Reach(SAS) \wedge \forall C \in C_{consist} : G \models C\}$ , the consistent states, consisting of all reachable states satisfying the consistency constraints,
- $Adapt(SAS) = \{G \mid G \in Reach(SAS) \wedge \exists C \in C_{adapt} : G \models C\}$ , the adaptation states, consisting of all reachable states satisfying some adaptation constraints,

- $Norm(SAS) = \{G \mid G \in Reach(SAS) \wedge \forall C \in C_{adapt} : G \not\models C\}$ , the normal states, consisting of all reachable states not satisfying any adaptation constraints.

For SA-systems, we require that

1. each pair of a context and a normal rule  $(p, r) \in R_{cont} \times R_{norm}$  is sequentially independent,
2. SAS is system consistent: all reachable states are consistent, i.e., they fulfill the consistency constraints:  $Reach(SAS) = Consist(SAS)$ ;
3. SAS is normal-state consistent: the initial state is normal and all normal rules preserve and reflect normal states:  
 $G_{init} \in Norm(SAS)$  and  $\forall G_0 \xRightarrow{r} G_1$  via  $r \in R_{norm}$   
 $[G_0 \in Norm(SAS) \Leftrightarrow G_1 \in Norm(SAS)]$
4. The set of adaptation rules  $R_{adapt}$  is confluent and terminating.

Figure 27 shows the *Reach* set, its partition into *Norm* and *Adapt* sets, and the relationships of these two sets with the *Consist* set. Note that the sets *Norm* and *Adapt* exclude each other.



**Fig. 27.** Relationships among the *Reach*, *Norm*, *Adapt*, and *Consist* sets.

*Remark 2.* - In all cases, the requirements of SA-systems can be concluded in a static way by inspecting the corresponding rules. This means e.g. that we do not need to check all reachable states to find whether they are consistent. Instead, we only check  $G_{init}$  for consistency and then check the (system or normal) rules whether they preserve consistent states. In particular, we can check statically that different adaptations do not interfere with each other, i.e., they are confluent and terminating (see requirement 4). This property is interesting, if more than one set of adaptation rules have to be used to adapt a given state, which is a highly relevant practical problem. Note that there are also general conditions ensuring the preservation of graph constraints by rules, but this discussion is out of scope for this paper.

*Example 8 (Car Logistics System as SA-system).*

We define the Car Logistics SA-system  $CLS = (GG, C_{sys})$  by the type graph  $TG$  in Figure 3, the initial state  $G_{init}$  in Figure 4, and the following sets of rules and constraints:

- $R_{norm} = \{\text{ServiceToDo}, \text{EnterBP}, \text{DoService}, \text{DoSub-Service}, \text{NextBA}, \text{FinishBP}\},$
- $R_{cont} = \{\text{SlightlyDamage}, \text{SeverelyDamage}, \text{Queue}\},$
- $R_{adapt} = R_{adapt}^1 \cup R_{adapt}^2 \cup R_{adapt}^3 \cup R_{adapt}^4$  with
- $R_{adapt}^1 = \{\text{Rollback}\},$
- $R_{adapt}^2 = \{\text{TakePullToTreatmentService}, \text{Take-RepairService}, \text{RepairVehicle}\}$
- $R_{adapt}^3 = \{\text{TakePullToTreatmentService}, \text{Take-RepairService}, \text{TakeDisposingService}, \text{RemoveTodo}, \text{RemoveDamageFlag}, \text{DisposeVehicle}\}$
- $R_{adapt}^4 = \{\text{Enqueue}, \text{DoService}, \text{RemoveWait}\},$
- $C_{consist} = \{\text{noFalseServiceConnect}, \text{sameBAfor-Comp}, \text{noEqualContextFlags}\},$
- $C_{adapt} = C_{adapt}^1 \cup C_{adapt}^2$  with
- $C_{adapt}^1 = \{\text{Damage}\},$
- $C_{adapt}^2 = \{\text{Wait}\}.$

The normal rules in  $R_{norm}$  and the context rules in  $R_{cont}$  have been shown and explained in Example 3. The context rules add context flags to vehicles or services, i.e., nodes of a certain Context subtype. These context nodes serve as indicators whether an adaptation becomes necessary or not (adaptation hooks). The adaptation rules in  $R_{adapt}^1$  to  $R_{adapt}^4$  have been introduced in Example 4 to Example 7, respectively. The consistency constraints in  $C_{consist}$  have been explained in Example 2 and model desired structural properties. The adaptation constraints in  $C_{adapt}$  model properties that have to be valid only if the corresponding adaptation is running. The adaptation constraints *Damage* and *Wait* (see Figure 5) require the existence of the corresponding adaptation hook.

Checking the requirements for SA-systems in Definition 3, we find that:

1. we have sequential independence for each pair of context and normal rules  $(p, r) \in R_{cont} \times R_{norm}$ , which is shown in the dependency matrix in Figure 28 computed by AGG. Each pair (SlightlyDamage, r), (SeverelyDamage, r) and (Queue, r) with  $r$  normal rule is sequentially independent (i.e., there are no dependencies for each rule pair, indicated by number 0 in the corresponding field of the table).

first \ second	1 ServiceP...2 EnterBP	3 DoService	4 DoSubS...	5 NextBA	6 FinishBP
1 SlightlyDamage	0	0	0	0	0
2 SeverelyDamage	0	0	0	0	0
3 Queue	0	0	0	0	0

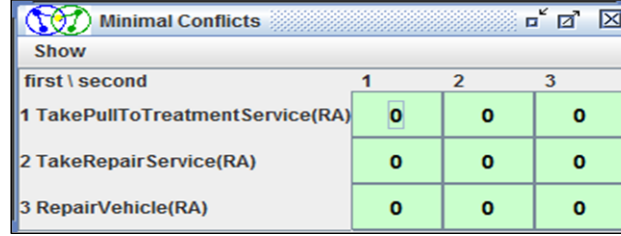
**Fig. 28.** Dependency matrix of context and normal rules

2.  $CLS$  is system consistent, because for all  $C \in C_{consist}$ ,  $G_{init} \models C$  and for all  $G_0 \xrightarrow{r} G_1$  via  $r \in R_{sys}$  and  $G_0 \in Consist(SAS)$  we also have  $G_1 \in Consist(SAS)$ . This can



be concluded since no system rules manipulate the structure of services; moreover, if a vehicle is linked to a service, it is checked by a NAC that this service is not a composite one (see rule `ServiceToDo` in Figure 8 ).

3.  $CLS$  is normal-state consistent, because  $G_{init} \in Norm(SAS)$  and for all  $G_0 \xrightarrow{r} G_1$  via  $r \in R_{norm}$  for all  $C \in C_{Adapt}$  [ $G_0 \not\models C \Leftrightarrow G_1 \not\models C$ ]. This can be concluded since no normal rule manipulates (inserts or deletes) any adaptation hooks (subtype of type `Context`), which is required by the adaptation constraints to hold.
4. With regard to confluence, we used AGG to check the sets of adaptation rules for critical pairs [26] (minimal conflicts) and found that there are no critical pairs, and hence, no conflicts for any rule pairs within the same adaptation rule set. Figure 29 shows exemplarily the critical pair table for the repair adaptation rule set, computed by AGG. We have listed the repair adaptation rules in the rows and columns and see number 0 in each field, denoting that there is no conflict for each rule pair.



first \ second	1	2	3
1 TakePullToTreatmentService(RA)	0	0	0
2 TakeRepairService(RA)	0	0	0
3 RepairVehicle(RA)	0	0	0

**Fig. 29.** Critical Pair matrix of repair adaptation rules

AGG computed some conflicts between different adaptation rule sets: there is e.g. a conflict when adaptation rule `Rollback` would be applied after rule `RepairVehicle`. Note that this conflict can be disregarded since the adaptation manager has to make sure to apply the rollback adaptation (if necessary) before evoking the repair adaptation and not afterwards. Similarly, conflicts between rules for repairing and rules for disposing vehicles can be ignored since we expect that in presence of severely damaged cars, the adaptation manager selects the *Dispose* adaptation first, and applies the *Repair* adaptation afterwards, when all severely damaged cars have been disposed of. Under these restrictions on the application order of adaptation rule sets, the union  $R_{adapt}$  of all adaptation rule sets is confluent.

With regard to termination, we argue as follows: The rollback adaptation  $R_{adapt}^1$  terminates as there are only a finite number of services to roll back within a composite service, and a `todo` edge may be inserted only once between a vehicle and a service. The repair adaptation  $R_{adapt}^2$  terminates due to a finite number of slightly damaged vehicles, and due to the NACs of the repair rules ensuring that each rule is applicable only once for each damaged vehicle. For the dispose-adaptation  $R_{adapt}^3$ , we have one rule, `RemoveDamageFlag` (Figure 22) that might be applicable at most twice, if a car has two `Damage` flags.

No car has more than two flags, due to NACs of the corresponding context rules. For the wait-at-queue adaptation  $R_{adapt}^4$ , we find that the NACs ensure that each vehicle is enqueued only once by rule **Enqueue** (Figure 24). The remaining rules only delete `todo` edges and are terminating due to the finite number of vehicles in the system. Hence, all adaptation rule sets are confluent and terminating.

## 5.2 Operational Properties of Self-Adaptive Systems

In this section, we define desirable operational properties of SA-Systems. One of the main ideas of SA-Systems is that they are monitored in regular time intervals and checked, whether the current system state is an adaptation state. In this case one or more adaptation hooks have been created in the last time interval by context rules. With the *enhancing*-adaptation property below, we require that a system in an adaptation state is eventually adapted, i.e., transformed again to a normal state, by adding new functionalities to the system and using new services (see section 3, where we discussed the *Nature of adaptations* on page 7). Moreover, *corrective* self-adaptation means that the state will be recovered, i.e., the normal state after adaptation is the same as if no adaptation had occurred. In the following, we use the notation  $G \Rightarrow^! G'$  to denote a transformation where the rules have been applied as long as possible; we write  $G \Rightarrow^* G'$  to denote a transformation where the rules have been applied arbitrarily often, and the transformation  $G \Rightarrow^+ G'$  consists of at least one rule application.

### Definition 4 (Self-Adaptation Classes).

An SA-System *SAS* is called

1. *enhancing*, if each adaptation state is adapted to become a normal state, possibly by adding new functionalities and services. In more detail:  
 $\forall G_{init} \Rightarrow^* G$  via  $(R_{norm} \cup R_{cont})$  with  $G \in Adapt(SAS) \exists G \Rightarrow^! G'$  via  $R_{adapt}$  with  $G' \in Norm(SAS)$  and  $\forall G \Rightarrow^! \bar{G}$  via  $R_{adapt}$ , we get that  $\bar{G} \cong G'$ .
2. *corrective*, if each adaptation state is adapted in a corrective way (repaired). In more detail:  
 $\forall G_{init} \Rightarrow^* G$  via  $(q_1 \dots q_n) \in (R_{norm} \cup R_{cont})^*$  with  $G \in Adapt(SAS) \exists G \Rightarrow^! G'$  via  $R_{adapt}$  with  $G' \in Norm(SAS)$  and  $\exists G_{init} \Rightarrow^* G'$  via  $(r_1 \dots r_m) \in R_{norm}^*$ , where  $(r_1 \dots r_m)$  is a subsequence of all normal rules in  $(q_1 \dots q_n)$ . Moreover,  $\forall G \Rightarrow^! \bar{G}$  via  $R_{adapt}$ , we get that  $\bar{G} \cong G'$ .

*Remark 3.* - By definition, each *corrective* SAS is also *enhancing*, but not vice versa. The additional requirement for corrective self-adaptation means that the system state  $G'$  obtained after adaptation is not only normal, but can also be generated by all normal rules in the given mixed sequence  $(q_1 \dots q_n)$  of normal and context rules, as if no context rule had been applied. We will see that our SA-System CLS is corrective, considering only the *Repair* adaptation for slightly damaged vehicles, but CLS together with the *Wait-At-Queue* adaptation would only be enhancing, but not corrective.

### 5.3 Analysis of Operational Properties

In this section, we analyze the operational properties introduced in subsection 5.2. We define direct, normal and rollback adaptation properties, which imply that the SAS is corrective / enhancing under suitable conditions in Theorem 1.

We want to ensure that for each context rule that adds an adaptation hook, there is a suitable adaptation grammar containing one or more adaptation rules leading again to a state without this adaptation hook, even if they are not applied immediately after its occurrence but later when the context monitor reveals that the adaptation hook must be invoked. This means that other normal and context rules may have been applied already, before the occurrence of the adaptation hook is monitored. For this reason we require in Theorem 1 that each pair  $(p, r)$  of context rules  $p$  and normal rules  $r$  is sequentially independent. By the Local Church-Rosser theorem for algebraic graph transformation [19] (Theorem 5.12) sequential independence of  $(p, r)$  allows one to switch the corresponding direct derivations in order to prove Theorem 1. For the case with nested application conditions including NACs we refer to [21]. Moreover, the AGG tool can calculate all pairs of sequentially independent rules with NACs in a static way.

#### Definition 5 (Self-Adaptation (SA) Properties).

Let  $G_0$  be a reachable state in the SA-system SAS. SAS has the

1. direct adaptation property, if the adaptation can be performed directly, i.e.,  $\forall G_0 \xRightarrow{p} G_1$  via  $p \in R_{cont} \exists G_1 \Rightarrow^* G_0$  via  $R_{adapt}$
2. normal adaptation property, if the necessary adaptation can be performed up to normal transformations leading to a possibly different normal state that is reachable from the state before the adaptation hook was set, i.e.,  $\forall G_0 \xRightarrow{p} G_1$  via  $p \in R_{cont} \exists G_1 \Rightarrow^+ G_2$  via  $R_{adapt}$  s.t.  $\exists G_0 \Rightarrow^* G_2$  via  $R_{norm}$ ,
3. rollback adaptation property, if the necessary adaptation can be performed up to normal transformations leading to a possibly different normal state from which the state before the adaptation hook was set is reachable, i.e.,  $\forall G_0 \xRightarrow{p} G_1$  via  $p \in R_{cont} \exists G_1 \Rightarrow^+ G_2$  via  $R_{adapt}$  s.t.  $\exists G_2 \Rightarrow^* G_0$  via  $R_{norm}$ .

*Remark 4.* Note that the normal and rollback adaptation property only differ in the direction of  $G_0 \Rightarrow^* G_2$  and  $G_2 \Rightarrow^* G_0$  via  $R_{norm}$ . For the normal and the rollback adaptation properties, it is required that the adapted state  $G_2$  is related to the old state  $G_0$  by a normal transformation. The direct adaptation property implies both the normal and the rollback property using  $G_2 = G_0$ .

#### Theorem 1 (Self-Adaptation Classes and their SA Properties).

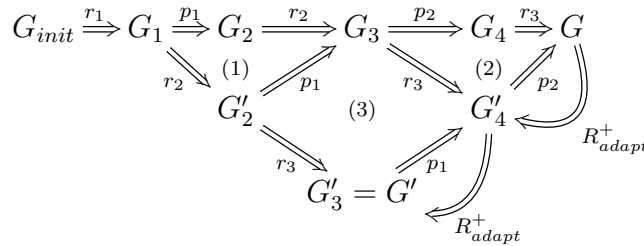
An SA-System SAS is

- I. corrective, if we have property 1 below
- II. enhancing, if we have
  - a) property 2 or b) properties 3 and 4 below.

1. *SAS has the direct adaptation property*
2. *SAS has the normal adaptation property.*
3. *SAS has the rollback adaptation property,*
4. *each pair  $(r, q) \in R_{norm} \times R_{adapt}$  is sequentially independent,*

*Proof.*

I. Given  $G_{init} \Rightarrow^* G$  via  $(q_1, \dots, q_n) \in (R_{norm} \cup R_{cont})^*$  with  $G \in Adapt(SAS)$  we have  $n \geq 1$ , because  $G_{init} \in Norm(SAS)$  since  $SAS$  is normal-state consistent. By sequential independence we can switch the order of  $(q_1, \dots, q_n)$ , s.t. first all normal rules  $r_i \in R_{norm}$  and then all context rules  $p_i \in R_{cont}$  are applied. As example let us consider  $G_{init} \Rightarrow^+ G$  via  $(r_1, p_1, r_2, p_2, r_3)$  with  $r_i \in R_{norm}$  and  $p_j \in R_{cont}$ . Then sequential independence leads by the Local Church-Rosser theorem to equivalent sequences in subdiagram (1), (2), (3) respectively.



By the direct adaptation property 1, we have  $G'_4 \Rightarrow^* G'_3$  and  $G \Rightarrow^* G'_4$  via  $R_{adapt}^*$ . With  $G' = G'_3$  we have  $G \Rightarrow^+ G'$  via  $R_{adapt}^*$  and  $G_{init} \Rightarrow^* G'$  via  $(r_1, r_2, r_3) \in R_{norm}^*$  where  $(r_1, r_2, r_3)$  is the subsequence  $(r_1, p_1, r_2, p_2, r_3)$  which consists of only normal rules, and normal-state consistency implies  $G_{init}, G' \in Norm(SAS)$ . Note that in the adaptation sequence  $G \Rightarrow^+ G'$  the (possible) adaptations due to the adaptation hooks caused by  $p_1, p_2 \in R_{cont}$  are performed in opposite order. In general, the sequence  $(q_1, \dots, q_n)$  ( $n \geq 1$ ) contains at least one rule in  $R_{cont}$ , because otherwise  $G \notin Adapt(SAS)$  (due to normal-state consistency), which is a contradiction to the assumption  $G \in Adapt(SAS)$ . This implies that we have an adaptation sequence  $G \Rightarrow^+ G'$  via  $R_{adapt}$ . Since adaptation rules are confluent and terminating, all possible adaptation transformations  $G \Rightarrow^+ \bar{G}$  lead to the same result  $\bar{G} \cong G'$ . Hence  $SAS$  is corrective.

II a) We can proceed as above up to the point that first all normal and then all context rules are applied. As shown in our example, the normal adaptation property 2 leads first to an adaptation transformation  $G \Rightarrow^+ G_5$  via  $R_{adapt}$  with  $G'_4 \Rightarrow^* G_5$  via  $R_{norm}$ , then we can switch rules in (4) according to sequential independence of context and normal rules. Finally the normal adaptation property 2. leads to  $G_5 \Rightarrow^+ G_6$  via  $R_{adapt}$  with  $G'_5 \Rightarrow^* G_6$  via  $R_{norm}$ .



from independent sources of disturbances issued by the environment. In our example, the context rules are all independent, i.e. if they are applicable in a sequence, their order can be swapped.

## 5.4 Verification of Operational Properties

In this section, we define static conditions for the self-adaptation properties defined in subsection 5.3. By Theorem 1, these static conditions are also sufficient conditions for the nature of our self-adaptive system. We then make use of these static conditions to verify the properties of the different adaptations of our Car Logistics Systems case study.

In Theorem 2 we give static conditions for the direct, normal and rollback adaptation properties. In part 1 of Theorem 2 we require that for each context rule  $p$  the *inverse rule*  $p^{-1}$  is *SAS-equivalent* to the *concurrent rule*  $q^*$  constructed from an adaptation rule sequence  $(q_1, \dots, q_n) \in R_{adapt}$ .

Let us explain shortly the notions *SAS-equivalent rules*, *inverse rule* and *concurrent rule* before stating Theorem 2.

*SAS-equivalent rules* model the same possible system changes:

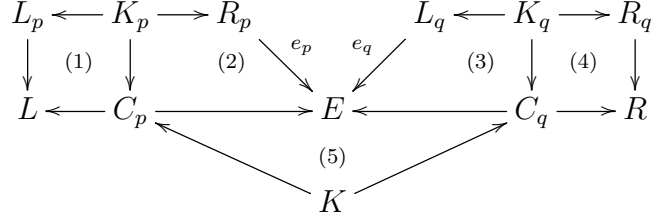
**Definition 6 (SAS-equivalent rules).** *Let  $SAS$  be an SA-system and  $r_1, r_2 \in R_{sys}$  be two system rules of  $SAS$ . Rules  $r_1$  and  $r_2$  are called SAS-equivalent (written  $r_1 \simeq r_2$ ) if  $(\exists G \xRightarrow{r_1} G') \iff (\exists G \xRightarrow{r_2} G')$  with  $G \in Reach(SAS)$ .*

*Remark 6 (Sufficient conditions for SAS-equivalent rules).* An obvious sufficient condition for checking SAS-equivalence of two rules is that the rules are isomorphic (two rules are isomorphic if they are componentwise isomorphic). Sometimes, this condition is too strong (as for our example as we will see later). A weaker sufficient condition is that the two rules are “isomorphic up to fixed objects”, i.e. one rule may contain more elements than the other rule under the condition that these additional elements are 1) preserved by the rule and 2) available in all states reachable by rules of the corresponding grammar. Moreover, 3) all NACs that are not isomorphic for both rules have to hold in all reachable states. Obviously, in this case the rules are applicable at the same matches and result in the same transformation. Note that conditions 1) to 3) can be checked statically by inspecting the initial state and the rules: for condition 2), we need to ensure that the required additional elements are present in the initial state, and are not removed by any system rule; analogously, for condition 3), we check that the elements forbidden by non-isomorphic NACs are not present in the initial state and are not added by any system rule.

For  $p = (L \leftarrow I \rightarrow R)$  with negative application condition  $nac : L \rightarrow N$  it is possible (see [19] Remark 7.21) to construct the inverse rule  $p^{-1} = (R \leftarrow I \rightarrow L)$  with equivalent  $nac' : R \rightarrow N'$ , such that  $G \xRightarrow{p} G'$  implies  $G' \xRightarrow{p^{-1}} G$  and vice versa.

A concurrent rule summarizes a given rule sequence in one equivalent rule [19,33]. In a nutshell, a concurrent rule  $p *_E q$  is constructed from two rules  $p = (L_p \leftarrow K_p \rightarrow R_p)$  and

$q = (L_q \leftarrow K_q \rightarrow R_q)$  that may be sequentially dependent via an overlapping graph  $E$  by the following diagram, where (5) is pullback and all remaining squares are pushouts:



The concurrent adaptation rule  $(p_1 * \dots * p_n)_E$  with  $E = (E_1, \dots, E_{n-1})$  for a longer sequence is constructed in an iterated way by  $(p_1 * \dots * p_n)_E = p_1 *_{E_1} p_2 *_{E_2} \dots *_{E_{n-1}} p_n$ . For the construction and corresponding properties of inverse and concurrent rules, which are needed in the proof of Theorem 2 we refer to [19].

In Theorem 2 we require as weaker conditions that each context rule  $p$  has a corresponding adaptation sequence  $(q_1 * \dots * q_n) \in R_{adapt}$ , which is not necessarily inverse to  $p$ . Note that we require that  $q$  is applicable after  $p$  has been applied, which is not a real static condition, but it can be argued from the context whether it is fulfilled. It is possible to define static conditions for applicability, but this is out of the scope of this paper. In part 2 of Theorem 2 it is sufficient to require for the normal adaptation property that we can construct a concurrent rule  $p *_{E_0} (q_1, \dots, q_n)_E$  which is *SAS*-equivalent to a concurrent rule  $r$  constructed from a normal rule sequence  $(r_1, \dots, r_m) \in R_{norm}$ . Analogously, for the rollback adaptation property we require that  $p *_{E_0} (q_1, \dots, q_n)_E$  is *SAS*-equivalent to an inverse concurrent normal rule  $r^{-1}$ .

### Theorem 2 (Verification of Self-Adaptation Properties).

Let *SAS* be an *SA*-system and  $G$  a reachable system state. *SAS* has

1. the direct adaptation property, if for each context rule  $p$  there is an adaptation rule sequence that directly reverses the effect of the context rule i.e.,  
 $\forall p \in R_{cont} \quad \exists q = (q_1 * \dots * q_n)_E$  via  $E = (E_1, \dots, E_{n-1})$  and  $n \geq 1, q_i \in R_{adapt}$  with  $q \simeq p^{-1}$ .
2. the normal (resp. rollback) adaptation property, if for each context rule  $p$  there is an adaptation rule sequence that reverses the effect of the context rule up to normal rule applications, i.e.,:  
 $\forall p \in R_{cont}$  we have
  - (a)  $\exists q = (q_1 * \dots * q_n)_E$  via  $E = (E_1, \dots, E_{n-1})$  and  $n \geq 1, q_i \in R_{adapt}$ , and  $q$  is applicable after  $p$  has been applied,
  - (b)  $\forall$  overlappings  $E_0$  of  $p$  and  $q$  leading to a concurrent rule  $p *_{E_0} q \quad \exists r = (r_1 * \dots * r_m)_{E'}$  with  $m \geq 1$  via  $E' = (E'_1, \dots, E'_{m-1})$  with  $r_i \in R_{norm}$  such that  $p *_{E_0} q \simeq r$  (resp.  $p *_{E_0} q \simeq r^{-1}$  in case of rollback).

*Proof.*

1. Given context rule  $p = (L \xleftarrow{l} K \xrightarrow{r} R)$  with NACs  $nac_{i,L} : L \rightarrow N_{i,L} (i \in I)$ , the inverse rule is given by  $p^{-1} = (R \xleftarrow{r} K \xrightarrow{l} L)$  with corresponding NACs  $nac_{i,R} : R \rightarrow N_{i,R}$ . Now, given  $p \in R_{cont}$  and  $G_0 \in Reach(SAS)$  with  $G_0 \xRightarrow{p} G_1$ , we have by assumption  $(q_1, \dots, q_n) \in R_{adapt}^*$  with  $(q_1 * \dots * q_n)_E \simeq p^{-1}$ , and by construction of  $p^{-1}$  also  $G_1 \xRightarrow{p^{-1}} G_0$  and hence also  $G_1 \xRightarrow{(q_1 * \dots * q_n)_E} G_0$  by *SAS*-equivalence, which implies by Concurrency Theorem  $G_1 \xRightarrow{*} G_0$  via  $R_{adapt}$ .

2. Given  $G_0 \in Reach(SAS)$  and  $G_0 \xRightarrow{p} G_1$  with  $p \in R_{cont}$ , by (a)  $\exists q = (q_1 * \dots * q_n)_E$  with  $n \geq 1$  and  $q_i \in R_{adapt}$  and  $q$  is applicable after  $p$  has been applied. Since  $G_0 \xRightarrow{p} G_1$  we also have  $G_1 \xRightarrow{q} G_2$  for some  $G_2$  and hence  $G_0 \xRightarrow{p} G_1 \xRightarrow{q} G_2$ . According to Fact 5.29 in [19],  $\exists$  overlapping  $E_0$  of  $p$  and  $q$  such that  $G_0 \xRightarrow{p} G_1 \xRightarrow{q} G_2$  are  $E_0$ -related and hence  $G_0 \xRightarrow{p *_{E_0} q} G_2$  by Concurrency Theorem [19,33].

By (b), we have  $\exists r = (r_1 * \dots * r_m)_{E'}$  with  $m \geq 1$  and  $r_i \in R_{norm}$  such that  $p *_{E_0} q \simeq r$  in case of normal adaptation property (resp.  $p *_{E_0} q \simeq r^{-1}$  in case of rollback adaptation property).

Hence, in case of normal adaptation property,  $G_0 \xRightarrow{p *_{E_0} q} G_2$  implies  $G_0 \xRightarrow{r} G_2$ . Now,  $G_0 \xRightarrow{r} G_2$  and  $G_1 \xRightarrow{q} G_2$  via concurrent rules  $r$  and  $q$  imply by Concurrency Theorem sequences  $G_0 \xRightarrow{+} G_2$  via  $R_{norm}$  and  $G_1 \xRightarrow{+} G_2$  via  $R_{adapt}$  leading to the normal adaptation property.

Similarly, in case of rollback adaptation property, we have that  $G_0 \xRightarrow{p *_{E_0} q} G_2$  implies  $G_0 \xRightarrow{r^{-1}} G_2$ , such that  $G_2 \xRightarrow{r} G_0$ .

Now,  $G_2 \xRightarrow{r} G_0$  and  $G_1 \xRightarrow{q} G_2$  via concurrent rules  $r$  and  $q$  with  $n, m \geq 1$  imply by Concurrency Theorem sequences  $G_2 \xRightarrow{+} G_0$  via  $R_{norm}$  and  $G_1 \xRightarrow{+} G_2$  via  $R_{adapt}$  leading to the rollback adaptation property.  $\square$

In the following examples 9 to 11, we verify the self-adaptation properties for different variants of our Car Logistics System case study *CLS* considering different adaptations. In Example 12 we give a counterexample, where the self-adaptation properties do not hold. In all four examples we have the same normal rules  $R_{norm} = \{\text{ServiceToDo}, \text{EnterBP}, \text{DoService}, \text{DoSubService}, \text{NextBA}, \text{FinishBP}\}$  (Figure 8 to 13) and a subset of the context rules  $R_{cont} = \{\text{SlightlyDamage}, \text{SeverelyDamage}, \text{Queue}\}$  (Figure 14 and 15) of *CLS*. Hence, we have sequential independence of context rules and normal rules, and, since *CLS* is normal-state consistent, also  $CLS_{Repair}$ ,  $CLS_{RollbackAndRepair}$ ,  $CLS_{Queue}$ , and  $CLS_{Dispose}$  are normal-state consistent.

*Example 9 (SA-System  $CLS_{Repair}$  is corrective).*

In  $CLS_{Repair}$ , we analyze the *Repair*-adaptation of *CLS*. This means, we have one context rule  $R_{cont} = \{\text{SlightlyDamage}\}$  (Figure 14), and the following set of adaptation rules:

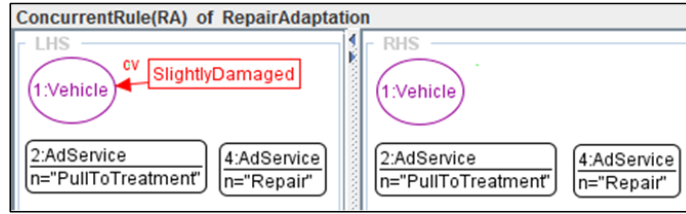


$R_{adapt} = R_{adapt}^2 = \{\text{TakePullToTreatmentService}, \text{TakeRepairService}, \text{RepairVehicle}\}$  (Figure 17 to 19). Moreover,  $C_{adapt} = \{\text{Damage}\}$  is the set of adaptation constraints.

According to Theorem 1, we have to show that  $CLS_{Repair}$  has the direct adaptation property (property 1). According to Theorem 2,  $CLS_{Repair}$  has the direct adaptation property if for  $p = \text{SlightlyDamage}$  we have  $(q_1, \dots, q_n) \in R_{adapt}$  with  $q = (q_1 * \dots * q_n)_E \simeq p^{-1}$ , where  $q$  is the concurrent rule of the adaptation rule sequence  $(q_1, \dots, q_n)$ .

This means, we have to find an adaptation rule sequence that results in a concurrent rule  $q$  which is *SAS*-equivalent to the inverse context rule  $p = \text{SlightlyDamage}$  (i.e., it removes the *SlightlyDamaged* flag).

We consider the adaptation rule sequence  $s = \{\text{TakePullToTreatmentService}, \text{TakeRepairService}, \text{RepairVehicle}\}$  together with suitable dependencies (overlapping) of the right-hand side of  $q_i$  and the left-hand side of  $q_{i+1}$ , and construct a concurrent rule from this sequence in an iterated way. In AGG, the construction of concurrent rules from rule sequences can be computed automatically. For our sequence, we get the concurrent adaptation rule shown in Figure 30.



**Fig. 30.** Concurrent Adaptation Rule  $q$  constructed from sequence  $s$  in  $CLS_{Repair}$

The concurrent adaptation rule  $q$  is *SAS*-equivalent to the inverse context rule  $p = \text{SlightlyDamage}$  due to the following argumentation: The additional elements in rule  $q$  wrt. rule  $p$  (the *PullToTreatment* and *Repair* nodes) are preserved by rule  $q$ , and are always there in all possible states, since no system rule ever adds or deletes *PullToTreatment* and *Repair* nodes.

Hence,  $CLS_{Repair}$  has the direct adaptation property, and due to Theorem 1, we can conclude that  $CLS_{Repair}$  is corrective.

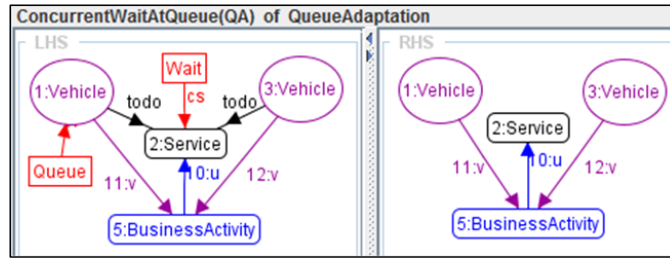
*Example 10 (SA-System  $CLS_{Queue}$  is enhancing).*

In  $CLS_{Queue}$ , we consider the *Wait-At-Queue*-adaptation. This means, we have one context rule  $R_{cont} = \{\text{Queue}\}$  (Figure 15), and the following set of adaptation rules:  $R_{adapt} = \{\text{Enqueue}, \text{DoService}, \text{RemoveWait}\}$  (Figure 24 to 26). Moreover,  $C_{adapt} = \{\text{Wait}\}$  is the set of adaptation constraints.

According to Theorem 1, we show that  $CLS_{Queue}$  has the normal adaptation property (property 2). According to Theorem 2,  $CLS_{Queue}$  has the normal adaptation property if for  $p = \text{Queue}$  we have

- (a)  $\exists q = (q_1 * \dots * q_n)_E$  via  $E = (E_1, \dots, E_{n-1})$  and  $n \geq 1, q_i \in R_{adapt}$  and  $q$  is applicable after  $p$  has been applied,
- (b)  $\forall$  overlappings  $E_0$  of  $p$  and  $q$  leading to a concurrent rule  $p *_{E_0} q \exists r = (r_1 * \dots * r_m)_{E'}$  via  $E' = (E'_1, \dots, E'_{m-1})$  with  $r_i \in R_{norm}$  such that  $p *_E q \simeq r$ .

For the context rule  $p = \text{Queue}$ , we have a sequence of adaptation rules  $s = \{\text{Enqueue}, \text{DoService}, \text{RemoveWait}\}$  that results in the concurrent rule  $q$  shown in Figure 31. Since the right-hand side of  $p$  equals the left-hand side of  $q$ , we can conclude that  $q$  is applicable after  $p$  has been applied as required in (a).



**Fig. 31.** Concurrent Adaptation Rule  $q$  constructed from sequence  $s$  in  $CLS_{Queue}$

We then construct according to (b) the concurrent rule  $\text{Queue} *_E q$  which equals rule  $q$  in Figure 31 but does not contain the context nodes `Wait` and `Queue` and their adjacent edges. Obviously, this rule  $\text{Queue} *_E q$  is isomorphic to a concurrent rule  $r$  constructed by the sequence of normal rules  $(\text{DoService} *_{E'} \text{DoService})$ , which removes two `todo` edges from two different vehicles to the same service in one step. Hence,  $\text{Queue} *_E q$  and  $r$  are *SAS*-equivalent.

Thus,  $CLS_{Queue}$  has the normal adaptation property, and due to Theorem 1, we can conclude that  $CLS_{Queue}$  is enhancing.

*Example 11* ( $CLS_{RollbackAndRepair}$  is enhancing).

In  $CLS_{RollbackAndRepair}$ , we consider the *Rollback-And-Repair*-adaptation. This means, we have the same context rule as in  $CLS_{Repair}$ , indicating that a vehicle becomes damaged. We also have the same set of adaptation rules together with one additional adaptation rule *Rollback* (Figure 16) which realizes the rollback actions of a composite service being currently performed on the damaged vehicle. In this case, all `todo` edges to service nodes belonging to the composite service are added again, and after the rollback-and-repair adaptation the repaired car has to start the composite service again from its beginning.

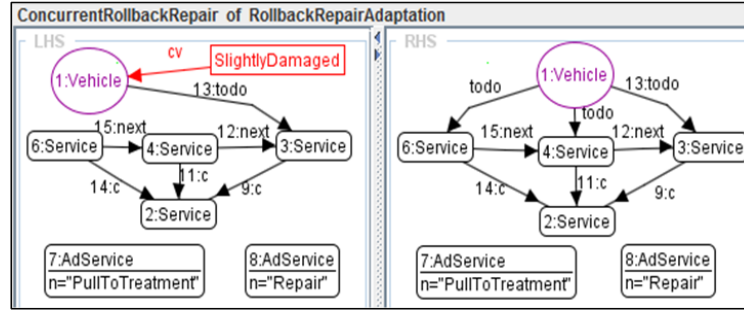
$R_{cont} = \{\text{SlightlyDamage}\}$ , and we have the following set of adaptation rules:  $R_{adapt} = \{\text{Rollback}\} \cup R_{adapt}^2$ . Moreover,  $C_{adapt} = \{\text{Damage}\}$  is the set of adaptation constraints.

According to Theorem 1, we show that 3)  $CLS_{RollbackAndRepair}$  has the rollback adaptation property, and that 4) each pair  $(r, q) \in R_{norm} \times R_{adapt}$  is sequentially independent.

ad. 3) According to Theorem 2,  $CLS_{RollbackAndRepair}$  has the rollback adaptation property if for  $p = \text{SlightlyDamage}$  we have

- (a)  $\exists q = (q_1 * \dots * q_n)_E$  via  $E = (E_1, \dots, E_{n-1})$  and  $n \geq 1, q_i \in R_{adapt}$  and  $q$  is applicable after  $p$  has been applied,
- (b)  $\forall$  overlappings  $E_0$  of  $p$  and  $q$  leading to a concurrent rule  $p *_{E_0} q \exists r = (r_1 * \dots * r_m)_{E'}$  via  $E' = (E'_1, \dots, E'_{m-1})$  with  $r_i \in R_{norm}$  such that  $p *_{E_0} q \simeq r^{-1}$ .

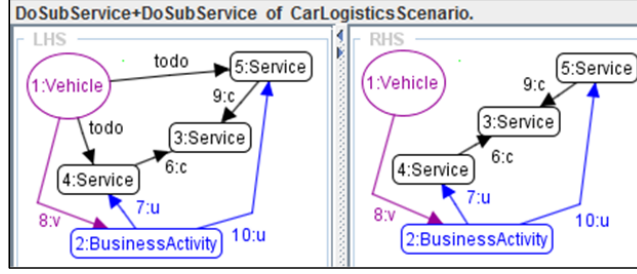
For the context rule **SlightlyDamage**, that occurs when the damaged car is half-way through a composite service, we have e.g., the sequence of adaptation rules  $s = \{\text{Rollback}, \text{Rollback}, \text{TakePullToTreatmentService}, \text{TakeRepairService}, \text{RepairVehicle}\}$  (see Figure 16 to 19) that results in the concurrent rule  $q$  shown in Figure 32. Note that if the composite service contains more than two subservices that have to be rolled back, then we need more **Rollback** rule instances in the beginning of our sequence  $s$ . Since the *RHS* of  $p$  is a subgraph of the *LHS* of  $q$  where  $q$  also requires that the damaged vehicle is in the midst of a composite service, we can conclude that *in this situation*  $q$  is applicable after  $p$  as required in (a).



**Fig. 32.** Concurrent rule  $q$  for adaptation rule sequence  $s$  with rollback adaptation of two subservices and subsequent repair adaptation in  $CLS_{RollbackAndRepair}$

For our case of two rollback steps, we get the concurrent rule  $\text{SlightlyDamage} *_{E_0} q$  which equals rule  $q$  in Figure 32 but does not contain the context node **SlightlyDamage** and its adjacent edge. Let us compare this rule  $\text{SlightlyDamage} *_{E_0} q$  with the inverse rule  $r^{-1}$  where  $r$  is the concurrent rule shown in Figure 33, that is constructed from the sequence of two normal rules ( $\text{DoSubService} *_{E'} \text{DoSubService}$ ). Hence,  $r^{-1}$  adds the two **todo** edges from the vehicle to two subservices in one step. Note that for  $n$  subservices, we need  $n$  rollback steps in the adaptation process and  $n$  applications of rule **DoSubService** in the concurrent rule  $r$ .

We see that rules  $\text{SlightlyDamage} *_{E_0} q$  and the inverse normal rule  $r^{-1}$  are *SAS*-equivalent due to the following: both rules add two **todo** edges from the vehicle to two different subservices, but rule  $\text{SlightlyDamage} *_{E_0} q$  additionally requires that there is at least one more subservice to do by the vehicle (i.e., a **todo** edge exists from the vehicle to a third subservice later in the composite service). Since this is exactly the situation to be expected for the adaptation *Rollback-and-Repair* and not changed by the rules, we get that  $\text{SlightlyDamage} *_{E_0} q$



**Fig. 33.** Concurrent rule  $r$  for the normal rule sequence  $(\text{DoSubService} *_{E'} \text{DoSubService})$  in  $CLS_{\text{RollbackAndRepair}}$

and the inverse normal rule  $r^{-1}$  are *SAS*-equivalent.

Hence,  $CLS_{\text{RollbackAndRepair}}$  has the rollback adaptation property.

ad. 4) The dependency matrix in Figure 34 computed by AGG shows that each pair  $(r, q) \in R_{\text{norm}} \times R_{\text{adapt}}$  is sequentially independent.

Minimal Dependencies					
Show					
first \ second	1 RollBack(...	2 TakePullT...	3 TakeRepai...	4 RepairVeh...	
1 ServicePlan.ServiceToDo	0	0	0	0	
2 EnterBP	0	0	0	0	
3 DoService	0	0	0	0	
4 DoSubService	0	0	0	0	
5 NextBA	0	0	0	0	
6 FinishBP	0	0	0	0	

**Fig. 34.** Dependency matrix of normal rules (rows) and adaptation rules (columns) in  $CLS_3$

With Theorem 1, we can conclude that  $CLS_{\text{RollbackAndRepair}}$  is enhancing.

*Example 12* ( $CLS_{\text{Dispose}}$  does not satisfy the sufficient conditions for SA properties).

In  $CLS_{\text{Dispose}}$ , we present a counter-example where our sufficient conditions showing that the SAS is enhancing are *not* satisfied. We consider the *Dispose-Vehicle*-adaptation, where a vehicle is severely damaged and has to be disposed. This means, we have one context rule ( $R_{\text{cont}} = \{\text{SeverelyDamage}\}$ , see Figure 14). We have the dispose adaptation rules  $R_{\text{adapt}}^3 = \{\text{TakePullToTreatmentService}, \text{TakeRepairService}, \text{TakeDisposingService}, \text{RemoveToDo}, \text{RemoveDamageFlag}, \text{DisposeVehicle}\}$  (Figure 17 to 23). Moreover,  $C_{\text{adapt}} = \{\text{Damage}\}$  is again the set of adaptation constraints.

According to Theorem 1, for  $CLS_{\text{Dispose}}$  to be enhancing, we should show that either 2)  $CLS_{\text{Dispose}}$  has the normal adaptation property, or 3)  $CLS_{\text{Dispose}}$  has the rollback adaptation

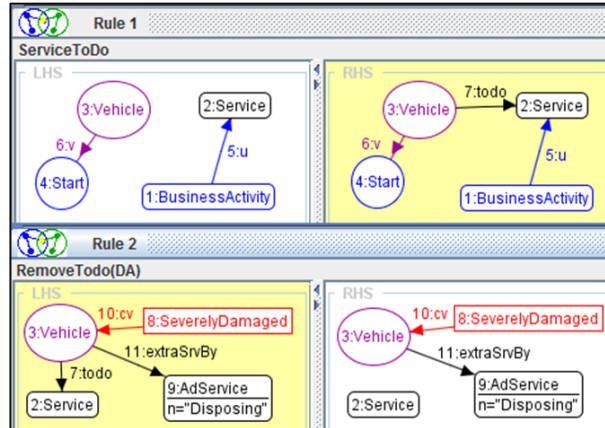
property, and 4) each pair  $(r, q) \in R_{norm} \times R_{adapt}$  is sequentially independent.

The dependency table in Figure 35 reveals that we do not have sequential independence for each pair  $(r, q) \in R_{norm} \times R_{adapt}$ , and hence condition 4) in Theorem 1 for  $CLS_{Dispose}$  to be enhancing is violated.

first \ second	1 TakePul...	2 TakeRe...	3 TakeDis...	4 Remove...	5 Remove...	6 Dispose...
1 ServicePlan.ServiceToDo	0	0	0	1	0	0
2 EnterBP	0	0	0	0	0	0
3 DoService	0	0	0	0	0	0
4 DoSubService	0	0	0	0	0	0
5 NextBA	0	0	0	0	0	0
6 FinishBP	0	0	0	0	0	0

**Fig. 35.** Dependency matrix of normal rules (rows) and adaptation rules (columns) in  $CLS_{Dispose}$

The reason for the dependency is that the adaptation rule **RemoveToDo** removes a **todo** edge that is inserted by normal rule **ServiceToDo**, as shown in the AGG detail view for the dependency in Figure 36.



**Fig. 36.** Dependency of rule **ServiceToDo** and rule **RemoveToDo** in  $CLS_{Dispose}$

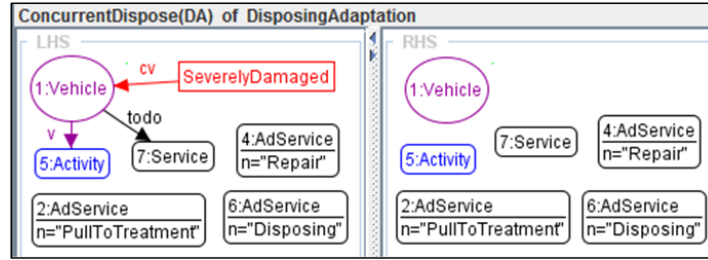
Alternatively, if we could show property 2), i.e., that  $CLS_{Dispose}$  has the normal adaptation property,  $CLS_{Dispose}$  would be proved enhancing anyway. According to Theorem 2,

$CLS_{Dispose}$  has the normal adaptation property if for  $p = \text{SeverelyDamage}$  (Figure 14) we have

- (a)  $\exists q = (q_1 * \dots * q_n)_E$  via  $E = (E_1, \dots, E_{n-1})$  and  $n \geq 1, q_i \in R_{adapt}$ , and  $q$  is applicable after  $p$  has been applied,
- (b)  $\forall$  overlappings  $E_0$  of  $p$  and  $q$  leading to a concurrent rule  $p *_{E_0} q \exists r = (r_1 * \dots * r_m)_{E'}$  via  $E' = (E'_1, \dots, E'_{m-1})$  with  $r_i \in R_{norm}$  such that  $p *_{E_0} q \simeq r$ .

For the context rule  $p = \text{SeverelyDamage}$ , we take the sequence of adaptation rules  $s_4 = (\text{TakePullToTreatmentService}, \text{TakeRepairService}, \text{TakeDisposingService}, \text{RemoveTodo}, \text{RemoveDamageFlag}, \text{DisposeVehicle})$  that results in the concurrent rule  $q$  shown in Figure 37.

Since the right-hand side of  $p$  is included in the left-hand side of  $q$ , and the additional context (the vehicle is connected to an activity and has at least one more service to do) can be assumed to be present in the situation we want to apply this adaptation, we can conclude that  $q$  is applicable after  $p$  has been applied, as required in (a).



**Fig. 37.** Concurrent Adaptation Rule  $q$  for Dispose-adaptation constructed from sequence  $s_4$  in  $CLS_{Dispose}$

We then construct according to (b) the concurrent rule  $\text{SeverelyDamage} *_{E_0} q$  which equals rule  $q$  in Figure 37 but does not contain the context node  $\text{SeverelyDamage}$  and its adjacent edge. Obviously, this rule  $\text{SeverelyDamage} *_{E_0} q$  is *not* SAS-equivalent to any concurrent rules constructed by the sequence of normal rules since we have no normal rules that delete an edge from a Vehicle node to an Activity node.

Hence,  $CLS_{Dispose}$  cannot be shown to be enhancing (in fact, due to our argumentation, we can be sure that it is *not* enhancing since we definitely do not find a normal rule sequence that generates the situation after the adaptation). By Remark 3, we can conclude that  $CLS_{Dispose}$  is also not corrective.

## 6 Automating the approach

AGG [46,43] is a well-established tool environment for algebraic graph transformation systems. Applications of AGG include graph and rule-based modeling of software, validation

of system properties by assigning an operational semantics to some system models, graph transformation-based evolution of software, and the definition of visual languages by graph grammars.

Graphs in AGG are defined by a type graph with node type inheritance and may be attributed by any kinds of Java objects. Graph transformations can be equipped with arbitrary computations on these Java objects described by Java expressions. The AGG environment consists of a graphical user interface comprising several visual editors, an interpreter, and a set of validation tools. The interpreter allows the stepwise transformation of graphs as well as rule applications as long as possible. The selection of the next applicable rule, as well as the selection of one possible match is implemented as being non-deterministic to adhere to the theory of graph transformation [19]. AGG supports several kinds of validations which comprise graph parsing, consistency checking of graphs, applicability checking of rule sequences, and conflict and dependency detection by critical pair analysis [26] of graph rules. Furthermore, checking the applicability of rule sequences, and constructing concurrent rules from rule sequences based on *object flow* is supported, as well. Object flow between rules has been defined in [30] as partial rule dependencies relating nodes of the RHS of one rule to (type-compatible) nodes of the LHS of a (not necessarily direct) subsequent rule in a given rule sequence. Object flow thus enhances the expressiveness of graph transformation systems and reduces the match finding effort. In AGG, object flow can be defined between subsequent rules in a rule sequence, and the rule sequence can be applied to a given graph respecting the object flow [43].

As shown in the previous sections, our framework for modeling and analyzing SA-systems is supported by the AGG tool to model both the initial configuration of the system and also the possible configurations that the system can reach in case of adaptations. A simulation of adaptations is performed by applying adaptation rules within AGG so that practitioners can easily and profitably get confidence on the system and on its evolutions.

From the modeling point of view, referring to the example, within AGG the engineer can perform the system design in an easy and direct way: as can be seen, the business process and services shown in Figure 1 can be directly mapped to elements of the initial state graph shown in Figure 4. Moreover, as can be seen in section 4, even graph constraints can be graphically represented. The behavior and the evolution of the system is also graphically represented within AGG in an intuitive way.

From the analysis point of view, dependencies between rules and conflicts between rules in a minimal context (critical pairs) can be computed fully automatically. The results support our argumentation showing that the sufficient conditions for our two theorems are satisfied in our examples. Moreover, the construction of concurrent rules from rule sequences is also fully automatic. It is only required that the modeler defines a suitable object flow between the rules of the sequence to define the overlapping graphs of rules and to get a unique resulting concurrent rule.

In the following, we discuss some aspects to assess the applicability of the proposed analysis techniques.

- **Practical relevance of assumptions:** For the Car Logistics case study, we found the assumptions for SA-systems very helpful for structuring the model (in particular by distinguishing different sets of rules for normal behavior, context changes and different adaptations). The sufficient conditions we had to check for applying our results did not prove to be too strong. Instead, whenever we found that our model did not satisfy one of the sufficient conditions, the changes we implemented in the model did not only result in the satisfaction of the conditions but also in a more systematic and concise model.
- **Achieved degree of automation:** All our analysis techniques are static, i.e., they check rule properties only. Yet, some of the techniques require manual effort, i.e., reasoning about rule properties that are not supported by AGG in a fully automatic way. For instance, although AGG implements checking sufficient conditions for termination of rule sets, for our example these sufficient conditions turned out to be too strong. So we had to argue about termination by inspecting the rules “manually”. Similarly, since up to now there is no automatic check for *SAS*-equivalence of rules implemented, we had to perform these checks by hand by inspecting the rules ourselves. Critical-pair analysis is a powerful instrument assisting with checking confluence of rule sets. A sufficient condition for the confluence of a system is (termination and) the absence of critical pairs. But if critical pairs are found, indicating potential conflicts, manual effort is needed to show whether these critical pairs could really lead to conflicts in a specific system or not. Usually, these hints are very helpful for the modeler and show where problems lie and the model should be adapted. Currently, the adaptation manager selects an adaptation grammar manually that is the most suitable for the present adaptation hooks. In the future, this decision making process could be supported by defining e.g., priorities for adaptation grammars depending on the severity of the disturbance (“treat worst case first”). This is currently not supported by AGG. Finding suitable adaptation sequences using an adaptation grammar is realized in a semi-automatic way by simulation: by applying the adaptation rules non-deterministically, the adaptation is performed automatically, and by keeping track manually of the used rules and their matches, the rule sequence and its object flow is determined. The object flow of the used rule sequence is then the input to AGG’s automatic construction of a concurrent adaptation rule. It would be desirable if for larger grammars, AGG could automatically record the rule application order and their matches when applying rules from the grammar. The recorded sequence and its object flow derived from the match overlappings could then be used to construct a concurrent rule fully automatically from a simulation run. Modeling the case study presented in our paper, we found modeling errors with the help of AGG which resulted in several improvement iterations on our model.



- **Time and memory consumption:** Figure 38 shows the time consumption for the fully automatic computation of dependency and conflict tables in our example<sup>7</sup>, depending on the number of rules in the corresponding grammar and the number of objects (nodes and edges) in the rules. We see that time and memory use increase exponentially for large rules. It is hence advisable to use more but smaller rules instead of describing the system by less larger rules.

Matrix	Rules	Nb. Rules	Nb. Objs	Time (ms)	Memory (k)
Dependency	(n, n)	6 x 6	5 - 9	5179	73453
Dependency	(c, n)	3 x 6	1 - 9	390	315
Dependency	(n, r)	6 x 3	5 - 6	795	8388
Dependency	(n, d)	6 x 6	5 - 8	1576	19161
Conflicts	(n, n)	6 x 6	5 - 9	4010	53737
Conflicts	(r, r)	3 x 3	4 - 6	539	9327
Conflicts	(d, d)	6 x 6	4 - 8	2343	33914
Conflicts	(q, q)	3 x 3	10 - 15	114768	44956

**Fig. 38.** Dependency and conflict analysis (*n*: normal, *c*: context, *r*: repair, *d*: dispose and *q*: queue rules)

Figure 39 shows the time consumption for the fully automatic computation of concurrent rules from rule sequences that were used in our example. For all concurrent rules, the time is below one second.

Concurrent Rule	Sequence size	Time (ms)	Memory (k)
RepairAdaptation	3	127	1099
DisposeAdaptation	6	937	5050
QueueAdaptation	3	983	1589

**Fig. 39.** Concurrent rules constructed from sequences

- **Scalability:** Obviously, time and memory consumption grows when the modeled system becomes larger. However, when the system becomes more complex w.r.t. the number of rules and the size of the system graphs, we find that the size of rules (modeling only local effects) remains nearly stable. It is in the hands of the modeler (adaptation manager) to formulate a rule set that can remain small enough to be analyzed properly. The size of the system graphs does not influence the performance of our static analyses. Up to now, AGG can analyze rules with up to 20-30 elements in reasonable time.
- **Addition of rules at run-time:** Since our framework for modeling SA-systems is modular, i.e., based on different rule sets, it is feasible and practical to add new rule sets

<sup>7</sup> measured on a standard notebook with Intel dual-core processor and 2 GB of memory

(adaptation grammars) to the system at runtime. To add a new adaptation hook to the system, the type graph must be updated by adding the new adaptation hook type. Ideally, this type is a subtype of a more general adaptation hook type (e.g. by adding a new AdaptV subtype *Special*, denoting that a car must undergo one more special service). In this case, the existing rules need not to be updated at all, only a new context rule and a new adaptation grammar specific to the new adaptation hook have to be added to the system. The adaptation manager realizes the selection of available adaptation grammars and this may include new grammars when needed.

## 7 Related Work

In this section, we compare our work with related approaches to adaptive system modeling and analysis. The comparison focuses on the aspects *context-awareness*, *self-adaptiveness*, *formalization* and *tool support* which we see as main aspects of self-adaptive system modeling.

Considering our own previous work, in [13] we proposed for the first time an approach to model self-repairing system architectures as typed (hyper) graph grammars and verified them w.r.t. dependencies and conflicts of rules modeling the environment, the normal system and repair actions. This approach was extended in [12] to model self-healing (SH) systems using algebraic graph transformation. SH systems are characterized by an automatic discovery of system failures, and by techniques to recover from these situations. In [12], preliminary sufficient conditions were formulated allowing for a static analysis of self-healing system properties. In our current paper we build on these preliminary results and extend the approach to the class of adaptive systems that is identified in section 2.

In comparison to our current paper, the static conditions in [12] were much more restrictive (requiring, e.g., a repair rule inverse to the context rule to ensure the so-called *direct-healing property*) Our current approach generalizes the approach in [12] with respect to the following aspects: concerning self-healing systems, we now allow for more general static conditions to ensure corrective behavior; for the *direct adaptation property*, we require the existence of a *sequence* of adaptation rules that reverses the effect of the context rule (up to normal rules) instead of a single inverse rule as in [12]. Furthermore, we do not restrict to self-healing systems anymore, but we now also consider systems that fall in the category of *enhancing* adaptive systems, i.e., systems that enhance existing services of the application, and whose adaptation states can be adapted to become normal again, but up to new functionalities or services. For ensuring that a system is indeed enhancing, we now check statically the *normal (resp. rollback) adaptation property*, requiring that there are adaptation rule sequences leading back to normal states (instead of looking only for inverse adaptation rules as in [12]).

The work in [11] presents a conceptual framework for adaptation. Through an interesting discussion on whether a system is adaptive or not, authors define adaptation as the runtime modification of the control data; control and data are two conceptual ingredients that compatibly with available resources determine the behavior of a component. It is important

to note that the work focuses on the adaptivity of simple components without considering adaptation that may emerge from the interactions among various components. Our approach is compatible with this conceptual framework, in the sense that the control data can be identified by the dividing of the set of rules in rules that correspond to ordinary computations and in rules that implement adaptation mechanisms.

Looking at different approaches to model self-adaptive systems, we find that although different software engineering techniques have been studied to deal with constantly changing landscapes, these techniques are mainly confined to specific research areas: software architectures [7,13,24], middleware [35,41], component-based development [37], service-oriented systems [15], etc.

Designing the adaptation of systems on the architecture level [38] offers several potential benefits, such as the appropriate abstraction level to describe dynamic changes in a system, the potential for scalability to large-scale complex applications, and the generality that allows one to design solutions for a wide range of application domains [32].

There is a wealth of Architecture Description Languages (ADLs) and architectural notations which provide support for dynamic software architectures analysis [9]. Some ADLs support the dynamic reconfiguration of the system Software Architecture (SA) taking advantage of Aspect-Oriented Software Development (AOSD) techniques [22].

Bencomo and Blair in [7] propose an approach called *Genie* that offers management of structural variability of adaptive systems. *Genie* can be considered as an ADL with generative capabilities to reconfigure from one system structure to another according to changes in the environment and to decide what kind of structural reconfiguration has to be performed. *Genie* uses domain specific languages for the construction of system models associated with both the structural (architectural) and the environmental variability. Using models and a middleware platform, each system can be dynamically reconfigured from one structure to another according to changes in the context (or environment) and is able to decide what kind of structural reconfiguration has to be performed. The main limit of this approach is the absence of a way to guarantee desired properties of the systems after each adaptation execution; in fact the language is not supported by any formal framework. From the modeling point of view the approach proposed in [7] is specifically architectural; we propose a general approach that can be used at different level of abstractions. For instance, our case study presents the business process of a service-oriented scenario.

Garlan et al. [24] introduce an SA-based self-adaptation framework, called *Rainbow*, which uses external mechanisms and an SA model to monitor a managed system, detect problems, determine a course of action, and carry out the adaptation actions. *Rainbow*, by making use of architectural styles, provides general and reusable infrastructures with explicit customization points. The definition of these customization points limits the dynamicity of the approach; in particular, the context is not considered as a part of the system model that can evolve during the system life-cycle. In our approach we do not rely on pre-defined customization points to manage the adaptation. Contrariwise, we monitor properties of the context to understand where and how adapt the system. As mentioned in [25], in order to

support services whose adaptive behavior can evolve over time, there is a need for higher level adaptation policies that are architecture independent.

Some years ago, Le Métayer [36] described software architectures in terms of graphs; an architecture style is seen as a set of architectures exhibiting a common shape. The dynamic adaptation of an architecture is defined by a coordinator who applies conditional graph rewriting rules. These rules are statically checked to ensure that the coordinator does not break the constraints of a given architecture style. Hirsch et al. [27] presented a similar approach for the specification of software architecture styles using hyperedge replacement systems. To model adaptation authors use graph rewriting combined with constraint solving; this allows to specify how components evolve and communicate. These papers posed important bases for formal modeling and verifying self-adaptive systems but, with respect to our approach, the limits are: (i) the way they check the system correctness, and (ii) the tool support. Regarding system correctness, they need to inspect all the reachable system states for each property that needs to be verified. Contrariwise, in our approach we define a set of operational properties (corrective, enhancing, direct (normal) adaptation, etc.) that we check in a static way by inspecting only the related rules without producing all reachable states explicitly. Regarding tool support, [36,27] provide a formal framework without existing tool support, whereas our formal framework is supported by AGG that is used to model and analyze self-adaptive systems.

Baresi et al. [4] present an approach to check whether an architecture is a refinement of another one. This is obtained by defining refinement relationships between abstract and concrete styles. The defined refinement criteria guarantee both semantic correctness and platform consistency. Refinement involves the reachability analysis for a target configuration from a given initial configuration. Reachability analysis is performed by using model checking and simulation. Model checking requires to a priori restrict the systems to finite state systems: this implies to fix an upper bound for the number of dynamic model elements that can be created by the transformation rules. Static analysis techniques, as applied in our paper, do not have this limitation. Moreover, we focus on modeling and verifying self-adaptive systems. We identified the class of self-adaptive systems we are able to deal with, and we provide a framework to verify consistency and operational properties of the self-adaptive systems concerning conflicts and dependencies of both normal system behavior, and corrective and enhancing adaptation.

Becker and Giese [6] present a graph transformation based approach to model correct self-adaptive systems on a high level of abstraction. The approach considers different levels of abstractions according to the reference architecture presented in [32]. The correctness of the modeled self-adaptive systems is checked by using simulation and invariant checking techniques. Invariant checking is mainly used to verify that a given set of graph transformations will never reach a forbidden state. However, the verification is efficient and the complexity is linear in the number of rules and properties to be checked [5]. The limitation of this approach is that they propose a unique model for both application and adaptation logics. This means that any time they need to add a new adaptation case, they need to refine the overall

model. In our approach the adaptation logic is developed separately from the application logic in terms of adaptation rules. In this way the adaptation logic is independent from the application, and then it can be modified without requiring changements on the application.

In the community of Service Oriented Computing, various approaches supporting self-healing have been defined, e.g., triggering repairing strategies as a consequence of a requirement violation [45], and optimizing QoS of service-based applications [14,49], or for satisfying some application constraints [47]. Repairing strategies could be specified by means of policies to manage the dynamism of the execution environment [3,18] or of the context of mobile service-based applications [42]. The goal of the strategies usually proposed by the aforementioned approaches range from service selection to rebinding and application reconfiguration [39]. All the previously mentioned techniques have interesting features, but even those that enable the definition of various adaptation strategies lack a coherent design approach to support designers in this complex task.

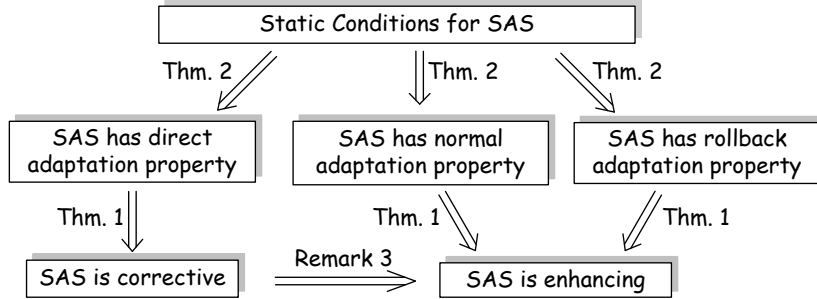
Summarizing, the approach that we propose in this paper abstracts from particular languages and notations. Moreover, our approach can be applied at different levels of granularity and it is not confined within the software architecture or service oriented computing domains. Instead, in this paper we provide a holistic and coherent design approach that allows software engineers to model and analyze self-adaptive systems within the same framework. Once a suitable level of abstraction being identified, the system can be modeled together with adaptation strategies and mechanisms. The system specification is then used to formally verify operational properties.

## 8 Conclusion and Future Work

In this paper, we have modeled and analyzed self-adaptive systems using algebraic graph transformation and graph constraints. We have defined consistency properties that include system consistency, normal state consistency, and adaptation state consistency. Moreover we defined operational properties that include self-adaptation, corrective self-adaptation and enhancing self-adaptation; we also defined direct, normal, and rollback adaptation properties concerning the behavior of adaptations w.r.t. their influence on the normal system behavior. Our analysis finds out in which class of self-adaptive systems a given system belongs (enhancing, corrective), and which properties we have with respect to the kind of adaptation (direct, normal, rollback). The classification helps to reason about system behavior, where e.g., systems with the rollback adaptation property may be more in danger of repeated failures than systems with normal adaptation property, since states that preceded failures are reached again after the adaptation. Note that the operational properties concern *all* reachable system states, whereas they are checked in our approach in a static way by inspecting only the rules without producing all reachable states explicitly.

The main results concerning operational properties are summarized in Figure 40, where most of the static conditions in Theorem 1 and Theorem 2 can be automatically checked by the AGG tool. We needed manual effort to show the termination of properties and *SAS*-

equivalence of rules, but it was always possible to perform the analysis statically. Although static conditions lead to over-approximating systems, we found that the conditions we had to check were reasonable to be expected to hold in SA-systems and did not restrict our intuitive notion of SA-system properties.



**Fig. 40.** Operational properties of self-adaptive systems

Exemplarily, the different properties are verified for different adaptations of our car logistics system in a seaport terminal.

In our approach, the selection of an adaptation grammar is done by a human adaptation manager who selects the most suitable adaptation in the case that more than one adaptation are possible. A distinguished kind of *control attributes* (like e.g., failure counters or timeout parameters) might be used in an extended approach to select automatically between different variants of adaptations. Note that states with additional control attributes (typed over control types) would still be *normal states* in our framework (not violating adaptation constraints). Our results hence can be extended in a straightforward way by reformulating the more flexible operational properties of enhancing and corrective systems to hold “up to changed control attributes”. In this case, a corrective SA system would no more be characterized by the fact that the adaptation returns to the normal state  $s$  before the failure, but by the fact that the adaptation returns to a normal state that is related to state  $s$  in a way that both states differ in the control elements only (formalized by a restriction of a system state graph to a view that does not contain the control types). Obviously, all operational properties and proofs have to be extended by checks for isomorphic restrictions of graphs. In this paper, we refrained from this extension to keep proofs clear and simple.

Work is in progress to evaluate the usability of our approach by applying it to larger case studies, and to further automate the checks currently needing manual effort with AGG. To further enhance the practical usability of static analysis, a continuous optimization of the performance of the critical pair analysis AGG is ongoing work. Moreover, the implementation of a logging feature for recording the order and matches of applied rules in AGG would be very helpful to automate the selection of rule sequences in our adaptation framework. As future work, we will investigate how far the techniques in this paper can be used for dealing

with general unpredictable adaptability. Another future research direction is to investigate the use of the techniques presented in this paper to deal with systems that have various operational modes [28]; modes are commonly used in embedded systems. Finally, we plan an integration of our formal framework with a real process engine like JBoss jBPM [29]. The idea is to use our framework as an analysis tool to guarantee the system consistency when each adaptation is executed.

## References

1. Abowd, G.D., Dey, A.K., Brown, P.J., Davies, N., Smith, M., Steggles, P.: Towards a Better Understanding of Context and Context-Awareness. In: Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing (HUC '99). pp. 304–307 (1999)
2. Babaoglu, Ö., Jelasity, M., Montresor, A., Fetzer, C., Leonardi, S., van Moorsel, A.P.A., van Steen, M. (eds.): Self-star Properties in Complex Information Systems, Conceptual and Practical Foundations, Lecture Notes in Computer Science, vol. 3460. Springer (2005)
3. Baresi, L., Guinea, S., Pasquale, L.: Self-healing BPEL processes with Dynamo and the JBoss rule engine. In: ESSPE'07. pp. 11–20. ACM (2007)
4. Baresi, L., Heckel, R., Thöne, S., Varró, D.: Style-based modeling and refinement of service-oriented architectures. *Journal of Software and Systems Modeling (SOSYM)* 5(2), 187–207 (2005)
5. Becker, B., Beyer, D., Giese, H., Klein, F., Schilling, D.: Symbolic invariant verification for systems with dynamic structural adaptation. In: Int. Conf. on Software Engineering (ICSE). ACM Press (2006)
6. Becker, B., Giese, H.: Modeling of correct self-adaptive systems: A graph transformation system based approach. In: Soft Computing as Transdisciplinary Science and Technology (CSTST'08). pp. 508 – 516. ACM Press (2008)
7. Bencomo, N., Blair, G.S.: Using architecture models to support the generation and operation of component-based adaptive systems. In: Software Engineering for Self-Adaptive Systems. pp. 183–200 (2009)
8. Böse, F., Piotrowski, J., Scholz-Reiter, B.: Autonomously controlled storage management in vehicle logistics - applications of RFID and mobile computing systems. *International Journal of RT Technologies: Research an Application* 1(1), 57–76 (2009)
9. Bradbury, J.S., Cordy, J.R., Dingel, J., Wermelinger, M.: A survey of self-management in dynamic software architecture specifications. In: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems (WOSS '04). pp. 28–33. ACM (2004)
10. Brun, Y., Marzo Serugendo, G., Gacek, C., Giese, H., Kienle, H., Litoiu, M., Müller, H., Pezzè, M., Shaw, M.: Engineering self-adaptive systems through feedback loops. *Software Engineering for Self-Adaptive Systems* pp. 48–70 (2009)

11. Bruni, R., Corradini, A., Gadducci, F., Lluch-Lafuente, A., Vandin, A.: A conceptual framework for adaptation. In: Fundamental Approaches to Software Engineering - 15th International Conference, FASE 2012. pp. 240–254 (Tallinn, Estonia, March 24 - April 1, 2012)
12. Bucchiarone, A., Ehrig, H., Ermel, C., Runge, O., Pelliccione, P.: Formal analysis and verification of self-healing systems. In: Fundamental Approaches to Software Engineering (FASE'10). LNCS, vol. 6013. Springer (2010)
13. Bucchiarone, A., Pelliccione, P., Vattani, C., Runge, O.: Self-repairing systems modeling and verification using AGG. In: Joint Working IEEE/IFIP Conference on Software Architecture (WICSA'09). pp. 181–190. IEEE (2009)
14. Canfora, G., Penta, M.D., Esposito, R., Villani, M.L.: An approach for qos-aware service composition based on genetic algorithms. In: Proceedings of the 2005 conference on Genetic and evolutionary computation (GECCO '05). pp. 1069–1075 (2005)
15. Cardellini, V., Casalicchio, E., Grassi, V., Lo Presti, F., Mirandola, R.: Qos-driven runtime adaptation of service oriented architectures. In: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC/FSE '09). pp. 131–140. ACM (2009)
16. Chapin, N., Hale, J.E., Kham, K.M., Ramil, J.F., Tan, W.G.: Types of software evolution and software maintenance. *Journal of Software Maintenance* 13, 3–30 (January 2001), <http://dl.acm.org/citation.cfm?id=371697.371701>
17. Cheng, B.H.C., Giese, H., Inverardi, P., Magee, J., de Lemos et al., R.: Software Engineering for Self-Adaptive Systems: A Research Road Map. In: Software Engineering for Self-Adaptive Systems. Dagstuhl Seminar Proceedings (2008)
18. Colombo, M., Nitto, E.D., Mauri, M.: Scene: A service composition execution environment supporting dynamic changes disciplined through rules. In: SERVICE-ORIENTED COMPUTING ICSOC 2006. pp. 191–202 (2006)
19. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS Monographs in Theor. Comp. Science (2006)
20. Ehrig, H., Engels, G., Kreowski, H.J.J., Rozenberg, G. (eds.): Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools. World Scientific (1999)
21. Ehrig, H., Habel, A., Lambers, L.: Parallelism and Concurrency Theorems for Rules with Nested Application Conditions. ECEASST 26 (2010)
22. Filman, R.E., Elrad, T., Clarke, S., Aksit, M. (eds.): Aspect-Oriented Software Development. Addison-Wesley, Boston (2005)
23. Ganek, A.G., Corbi, T.A.: The dawning of the autonomic computing era. *IBM Syst. J.* 42, 5–18 (January 2003), <http://dx.doi.org/10.1147/sj.421.0005>
24. Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer* 37(10), 46–54 (2004)



25. Gjørven, E., Eliassen, F., Lund, K., Eide, V.S.W., Staehli, R.: Self-adaptive systems: A middleware managed approach. In: Alexander Keller, J.P.M.F. (ed.) 2nd IEEE International Workshop on Self-Managed Networks, Systems & Services (SelfMan 2006). Springer (2006)
26. Hausmann, J.H., Heckel, R., Taentzer, G.: Detection of conflicting functional requirements in a use case-driven approach. In: ICSE 2002. pp. 105–115. ACM Press (2002)
27. Hirsch, D., Inverardi, P., Montanari, U.: Reconfiguration of software architecture styles with name mobility. In: Proceedings of the 4th International Conference on Coordination Languages and Models. pp. 148–163. COORDINATION '00, Springer-Verlag, London, UK (2000), <http://dl.acm.org/citation.cfm?id=647016.713435>
28. Jahanian, F., Mok, A.K.: Modechart: A specification language for real-time systems. *IEEE Trans. Softw. Eng.* 20, 933–947 (December 1994), <http://dx.doi.org/10.1109/32.368134>
29. JBoss: jBPM Engine. <http://www.jboss.org/jbpm>
30. Jurack, S., Lambers, L., Mehner, K., Taentzer, G., Wierse, G.: Object Flow Definition for Refined Activity Diagrams . In: Chechik, M., Wirsing, M. (eds.) *Proc. Fundamental Approaches to Software Engineering (FASE'09)*. vol. 5503, pp. 49–63. Springer (2009), <http://www.springerlink.com/content/bgmnx25w700j4767/?p=83ac5bfbce90462dbb453b2a95fed6e2&pi=3>
31. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. *Computer* 36, 41–50 (January 2003), <http://dx.doi.org/10.1109/MC.2003.1160055>
32. Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. In: 2007 Future of Software Engineering. pp. 259–268. FOSE '07, IEEE Computer Society, Washington, DC, USA (2007), <http://dx.doi.org/10.1109/FOSE.2007.19>
33. Lambers, L.: Certifying Rule-Based Models using Graph Transformation. Ph.D. thesis, Technische Universität Berlin (2009)
34. Lanese, I., Bucchiarone, A., Montesi, F.: A framework for rule-based dynamic adaptation. In: Trustworthy Global Computing - 5th International Symposium, TGC 2010, Munich, Germany, February 24–26, 2010, Revised Selected Papers. *Lecture Notes in Computer Science*, vol. 6084, pp. 284–300. Springer (2010)
35. Liu, H., Parashar, M., Member, S.: Accord: A programming framework for autonomic applications. *IEEE Transactions on Systems, Man and Cybernetics* 36, 341–352 (2005)
36. Métayer, D.L.: Describing software architecture styles using graph grammars. *IEEE Trans. Software Eng.* 24(7), 521–533 (1998)
37. Peper, C., Schneider, D.: Component engineering for adaptive ad-hoc systems. In: Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems (SEAMS '08). pp. 49–56. ACM (2008)
38. Perry, D., Wolf, A.: Foundations for the Study of Software Architecture. *SIGSOFT Softw. Eng. Notes* 17(4), 40–52 (1992)
39. Pfeffer, H., Linner, D., Steglich, S.: Dynamic adaptation of workflow based service compositions. In: Proceedings of the 4th international conference on Intelligent Computing:

- Advanced Intelligent Computing Theories and Applications - with Aspects of Theoretical and Methodological Issues (ICIC '08). pp. 763–774. Springer-Verlag (2008)
40. Rodosek, G.D., Geihs, K., Schmeck, H., Burkhard, S.: Self-healing systems: Foundations and challenges. In: Self-Healing and Self-Adaptive Systems. No. 09201 in Dagstuhl Seminar Proceedings, Germany (2009)
  41. Rouvoy, R., Barone, P., Ding, Y., Eliassen, F., Hallsteinsen, S.O., Lorenzo, J., Mamelli, A., Scholz, U.: Music: Middleware support for self-adaptation in ubiquitous and service-oriented environments. In: Software Engineering for Self-Adaptive Systems. pp. 164–182 (2009)
  42. Rukzio, E., Siorpaes, S., Falke, O., Hussmann, H.: Policy based adaptive services for mobile commerce. In: WMCS'05. IEEE Computer Society (2005)
  43. Runge, O., Ermel, C., Taentzer, G.: AGG 2.0 – New Features for Specifying and Analyzing Algebraic Graph Transformations. In: Schürr, A., Várro, D. (eds.) Symposium on Application of Graph Transformation with Industrial Relevance (AGTIVE'11). LNCS, Springer (2012), to Appear
  44. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.* 4, 14:1–14:42 (May 2009), <http://doi.acm.org/10.1145/1516533.1516538>
  45. Spanoudakis, G., Zisman, A., Kozlenkov, A.: A service discovery framework for service centric systems. In: Proceedings of the 2005 IEEE International Conference on Services Computing (SCC '05). pp. 251–259 (2005)
  46. TFS-Group, TU Berlin: AGG (2013), <http://www.tfs.tu-berlin.de/agg>
  47. Verma, K., Gomadam, K., Sheth, A.P., Miller, J.A., Wu, Z.: The METEOR-S approach for configuring and executing dynamic web processes. Tech. rep., University of Georgia, Athens (2005)
  48. White, S.R., Hanson, J.E., Whalley, I., Chess, D.M., Segal, A., Kephart, J.O.: Autonomic computing: Architectural approach and prototype. *Integr. Comput.-Aided Eng.* 13(2), 173–188 (2006)
  49. Zeng, L., Benatallah, B., Dumas, M., Kalagnanam, J., Sheng, Q.Z.: Quality driven web services composition. In: Proceedings of the 12th international conference on World Wide Web (WWW '03). pp. 411–421 (2003)