# Formal Analysis and Verification of Self-Healing Systems*

Hartmut Ehrig[1], Claudia Ermel[1], Olga Runge[1],
Antonio Bucchiarone[2], and Patrizio Pelliccione[3]

[1] Institut für Softwaretechnik und Theoretische Informatik
Technische Universität Berlin, Germany
{ehrig,lieske,olga}@cs.tu-berlin.de
[2] FBK-IRST, Trento, Italy
bucchiarone@fbk.eu
[3] Dipartimento di Informatica Università dell'Aquila, Italy
patrizio.pelliccione@di.univaq.it

**Abstract.** Self-healing (SH-)systems are characterized by an automatic discovery of system failures, and techniques how to recover from these situations. In this paper, we show how to model SH-systems using algebraic graph transformation. These systems are modeled as typed graph grammars enriched with graph constraints. This allows not only for formal modeling of consistency and operational properties, but also for their analysis and verification using the tool AGG. We present sufficient static conditions for self-healing properties, deadlock-freeness and liveness of SH-systems. The overall approach is applied to a traffic light system case study, where the corresponding properties are verified.

## 1 Introduction

The high degree of variability that characterizes modern systems requires to design them with runtime evolution in mind. Self-adaptive systems are a variant of fault-tolerant systems that autonomously decide how to adapt the system at runtime to the internal reconfiguration and optimization requirements or to environment changes and threats [1]. A classification of modeling dimensions for self-adaptive systems can be found in [2], where the authors distinguish *goals* (what is the system supposed to do), *changes* (causes for adaptation), *mechanisms* (system reactions to changes) and *effects* (the impact of adaptation upon the system). The initial four self-* properties of self-adaptive systems are self-configuration, self-healing[1], self-optimization, and self-protection [4]. Self-configuration comprises components installation and configuration based on some high-level policies. Self-healing deals with automatic discovery of system

* Some of the authors are partly supported by the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement 215483 (S-Cube) and the Italian PRIN d-ASAP project.

[1] Following [3] we consider self-healing and self-repair as synonymous.

D.S. Rosenblum and G. Taentzer (Eds.): FASE 2010, LNCS 6013, pp. 139–153, 2010.

failures, and with techniques to recover from them. Typically, the runtime behavior of the system is monitored to determine whether a change is needed. Self-optimization monitors the system status and adjusts parameters to increase performance when possible. Finally, self-protection aims to detect external threats and mitigate their effects [5].

In [6], Bucchiarone et al. modeled and verified dynamic software architectures and self-healing (SH-)systems (called self-repairing systems in [6]), by means of hypergraphs and graph grammars. Based on this work, we show in this paper how to formally model (SH-)systems by using algebraic graph transformations [7] 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 [7,8].

The main idea is to model SH-systems by typed graph grammars with three different kinds of system rules, namely normal, environment, and repair rules. Normal rules define the normal and ideal behavior of the system. Environment rules model all possible predictable failures. Finally, for each failure a repair rule is defined. This formalization enables the specification, analysis and verification of consistency and operational properties of SH-systems. More precisely, we present sufficient conditions for two alternative self-healing properties, deadlock-freeness and liveness of SH-systems. The conditions can be checked statically for the given system rules in an automatic way using the AGG[2] modeling and verification tool for typed attributed graph transformation systems.

Summarizing, the contribution of this paper is twofold: (i) we propose a way to model and formalize SH-systems; (ii) we provide tool-supported static verification techniques for SH-system models. The theory is presented by use of a running example, namely an automated traffic light system controlled by means of electromagnetic spires that are buried some centimeters underneath the asphalt of car lanes.

The paper is organized as follows: Section 2 motivates the paper comparing it with related work. Section 3 presents the setting of our running example. Section 4 introduces typed attributed graph transformation as formal basis to specify and analyze SH-systems. In Section 5 we define consistency and operational system properties. Static conditions for their verification are given in Section 6 and are used to analyze the behavior and healing properties of the traffic light system. We conclude the paper in Section 7 with a summary and an outlook on future work. For full proofs of the technical theorems and more details of our running example, the reader is referred to our technical report [9].

## 2 Motivation and Related Work

Focusing on modeling approaches for SH-systems, the *Software Architecture* approach (SA) [10], has been introduced as a high-level view of the structural organization of systems. Since a self-healing system must be able to change at runtime, *Dynamic Software Architectures* (DSAs) have shown to be very useful

[2] AGG (Attributed Graph Grammars): http://tfs.cs.tu-berlin.de/agg.

to capture SA evolution [11,12,13]. Aiming at a formal analysis of DSAs, different approaches exist, either based on graph transformation [6,14,15,16,17,18,19] or on temporal logics and model checking [20,21,22]. In many cases, though, the state space of behavioral system models becomes too large or even infinite, and in this case model checking techniques have their limitations. Note that static analysis techniques, as applied in this paper, do not have this drawback. In addition to graph transformation techniques, also Petri nets [23] offer static analysis techniques to verify liveness and safety properties. But in contrast to Petri nets, graph transformation systems are well suited to model also reconfiguration of system architectures which is one possible way to realize system recovery from failures in self-healing (SH-)systems.

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 [24], and optimizing QoS of service-based applications [25,26]. Repairing strategies could be specified by means of policies to manage the dynamism of the execution environment [27,28] or of the context of mobile service-based applications [29].

In [30], a theoretical assume-guarantee framework is presented to efficiently define under which conditions adaptation can be performed by still preserving the desired invariant. In contrast to our approach, the authors of [30] aim to deal with *unexpected* adaptations.

In contrast to the approaches mentioned above, we abstract from particular languages and notations. Instead, we aim for a coherent design approach allowing us to model important features of SH-systems at a level of abstraction suitable to apply static verification techniques.

## 3 Running Example: An Automated Traffic Light System

In an automated Traffic Light System (TLS), the technology is based upon electromagnetic spires that are buried some centimeters underneath the asphalt of car lanes. The spires register traffic data and send them to other system components. The technology helps the infraction system by making it incontestable. In fact, the TLS is connected to cameras which record videos of the violations and automatically send them to the center of operations. In addition to the normal behavior, we may have failures caused by a loss of signals between traffic light or camera and supervisor. For each of the failures there are corresponding repair actions, which can be applied after monitoring the failures during runtime. For more detail concerning the functionality of the TLS, we refer to [9].

The aim of our TLS model is to ensure suitable self-healing properties by applying repair actions. What kind of repair actions are useful and lead to consistent system states without failures? What kind of safety and liveness properties can be guaranteed? We will tackle these questions in the next sections by providing a formal modeling and analysis technique based on algebraic graph transformation and continue our running example in Examples 1 – 6 below.

## 4 Formal Modeling of Self-Healing Systems by Algebraic Graph Transformation

In this section, we show how to model SH-systems in the formal framework of algebraic graph transformation [7]. The main concepts of this framework which are relevant for our approach are typed graphs, graph grammars, transformations and constraints. Configurations of an SH-System are modeled by typed graphs.

**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* $s, t : E \to N$ *assigning to each edge* $e \in E$ *the* source $s(e) \in N$ *and* target $t(e) \in N$*.*

*A* graph morphism $f : G \to G'$ *is given by a pair of functions* $f = (f_N : N \to N', f_E : E \to E')$ *which is compatible with source and target functions.*

*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* $t : G \to TG$*, called* typing morphism *of* $G$*.* Morphisms $f : \overline{G} \to \overline{G'}$ of typed graphs *are graph morphisms* $f : G \to G'$ *which are compatible with the typing morphisms of* $G$ *and* $G'$*, 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 [7], as used in our running example.

*Example 1 (Traffic Light System).* The type graph $TG$ of our traffic light system $TLS$ is given in Fig. 1. The initial state is the configuration graph in Fig. 2 which is a $TG$-typed graph where the typing is indicated by corresponding names, and the attributes are attached to nodes and edges. The initial state shows two traffic lights (TL), two cameras, a supervisor, and a center of operations, but no traffic up to now.

The dynamic behavior of SH-systems is modeled by rules and transformations of a typed graph grammar in the sense of algebraic graph transformation [7].

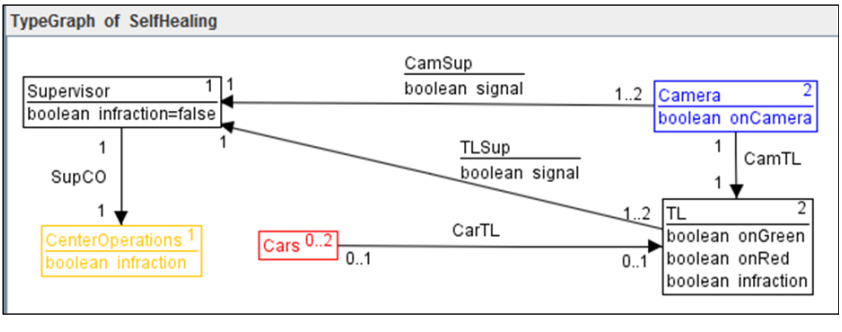
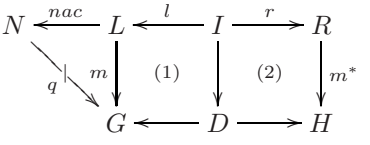



**Fig. 1.** TLS type graph $TG$

**Definition 2 (Typed Graph Grammar)**
*A* typed graph grammar $GG = (TG, G_{init}, Rules)$ *consists of a type graph* $TG$*, a* $TG$*-typed graph* $G_{init}$*, called* initial graph*, and a set Rules of* graph transformation rules*. Each rule* $r \in Rules$ *is given by a span* $(L \leftarrow I \rightarrow R)$*, where* $L, I$ *and* $R$ *are* $TG$*-typed graphs, called* left-hand side, right-hand side *and* interface*, respectively. Moreover,* $I \rightarrow L$*,* $I \rightarrow R$ *are injective typed graph morphisms where in most cases* $I$ *can be considered as intersection of* $L$ *and* $R$*. A rule* $r \in Rules$ *is applied to a* $TG$*-typed graph* $G$ *by a* match morphism $m : L \rightarrow G$ *leading to a* direct transformation $G \overset{r,m}{\Longrightarrow} H$ *via* $(r, m)$ *in two steps: at first, we delete the match* $m(L)$ *without* $m(I)$ *from* $G$ *to obtain a context graph* $D$*, and secondly, we glue together* $D$ *with* $R$ *along* $I$ *leading to a* $TG$*-typed graph* $H$*.*

*More formally, the direct transformation* $G \overset{r,m}{\Longrightarrow} H$ *is given by two* pushout diagrams (1) *and* (2) *in the category* $\mathbf{Graphs_{TG}}$ *of* $TG$*-typed graphs, where diagram* (1) *(resp.* (2)*) corresponds to gluing* $G$ *of* $L$ *and* $D$ *along* $I$ *(resp. to gluing* $H$ *of* $R$ *and* $D$ *along* $I$*).*




*Note that pushout diagram* (1) *in step 1 only exists if the match* $m$ *satisfies a* gluing condition *w.r.t. rule* $r$ *which makes sure that the deletion in step 1 leads to a well-defined* $TG-$*typed graph* $D$*. Moreover, rules are allowed to have* Negative Application Conditions *(NACs) given by a typed graph morphism* $nac : L \rightarrow N$*. In this case, rule* $r$ *can only be applied at match* $m : L \rightarrow G$ *if 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* $G_0 \overset{*}{\Longrightarrow} G_n$ *via Rules in* $GG$ *consists of* $n \geq 0$ *direct transformations* $G_0 \Longrightarrow G_1 \Rightarrow ... \Rightarrow G_n$ *via rules* $r \in Rules$*. For* $n \geq 1$ *we write* $G_0 \overset{+}{\Longrightarrow} G_n$*.*

*Example 2 (Rules of TLS).* A rule $r = (L \leftarrow I \rightarrow R)$ of $TLS$ with NAC $nac : L \rightarrow N$ is given in Fig. 3 (interface $I$ is not shown and consists of the nodes and edges which are present in both $L$ and $R$, as indicated by equal numbers). For simplicity, we only show the part of the NAC graph $N$ which extends $L$. All graph morphisms are inclusions. Rule $r$ can be applied to graph $G$ in Fig. 2 where the node (1:TL) in $L$ is mapped by $m$ to the upper node $TL$ in

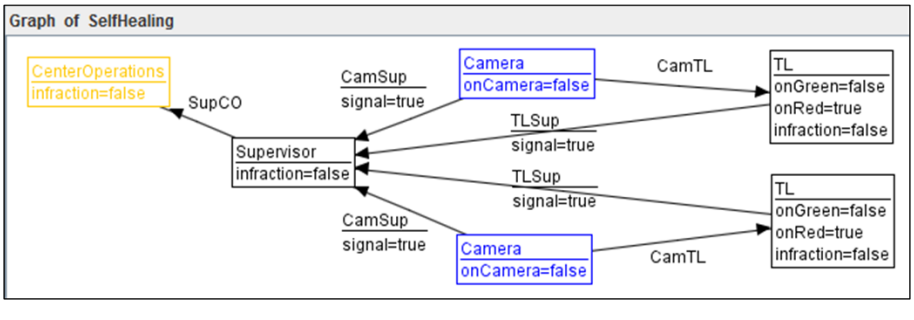



**Fig. 2.** TLS initial state $G_{init}$

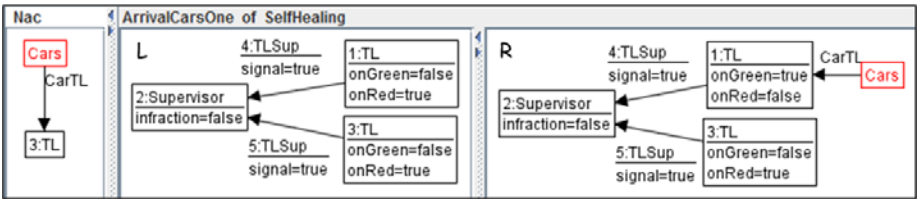



**Fig. 3.** TLS rule *ArrivalCarsOne*

$G_{init}$. This leads to a graph $H$ where the attributes of $TL$ are changed and the node Cars of $R$ is attached to $TL$. Altogether, we have a direct transformation $G \overset{r,m}{\Longrightarrow} H$.

In order to model consistency and failure constraints of an SH-system, we use graph constraints. A *TG-typed graph constraint* is given by a $TG$-typed graph $C$ which is *satisfied by a TG-typed graph* $G$, written $G \models C$, if there is an injective graph morphism $f : C \to G$. Graph constraints can be negated or combined by logical connectors (e.g. $\neg C$). Now we are able to define SH-systems in the framework of algebraic graph transformation (AGT). An SH-system is given by a typed graph grammar where four kinds of rules are distinguished, called *system*, *normal*, *environment* and *repair* rules. Moreover, we have two kinds of $TG$-typed graph constraints, namely *consistency* and *failure* constraints.

**Definition 3 (Self-healing System in AGT-Framework)**
*A Self-healing system (SH-system) is given by* $SHS = (GG, C_{sys})$, *where:*

- $GG = (TG, G_{init}, R_{sys})$ *is a typed graph grammar with type graph* $TG$, *a* $TG$*-typed graph* $G_{init}$, *called* initial state, *a set of* $TG$*-typed rules* $R_{sys}$ *with NACs, called* system rules, *defined by* $R_{sys} = R_{norm} \cup R_{env} \cup R_{rpr}$, *where* $R_{norm}$ *(called* normal rules*),* $R_{env}$ *(called* environment rules*) and* $R_{rpr}$ *(called* repair rules*) are pairwise disjoint.*
- $C_{sys}$ *is a set of* $TG$*-typed graph constraints, called* system constraints, *with* $C_{sys} = C_{consist} \cup C_{fail}$, *where* $C_{consist}$ *are called* consistency constraints *and* $C_{fail}$ failure constraints.

For an SH-system, we distinguish *reachable*, *consistent*, *failure* and *normal* states, where reachable states split into normal and failure states.

**Definition 4 (Classification of SH-System States)**
*Given an SH-system* $SHS = (GG, C_{sys})$ *as defined above, we have*

1. $Reach(SHS) = \{G \mid G_{init} \overset{*}{\Longrightarrow} G$ *via* $R_{sys}\}$, *the* reachable states *consisting of all states reachable via system rules,*
2. $Consist(SHS) = \{G \mid G \in Reach(SHS) \wedge \forall C \in C_{consist} : G \vDash C\}$, *the* consistent states, *consisting of all reachable states satisfying the consistency constraints,*
3. $Fail(SHS) = \{G \mid G \in Reach(SHS) \wedge \exists C \in C_{fail} : G \vDash C\}$, *the* failure states, *consisting of all reachable states satisfying some failure constraint,*

4. $Norm(SHS) = \{G \mid G \in Reach(SHS) \wedge \forall C \in C_{fail} : G \not\models C\}$, *the* normal states*, consisting of all reachable states not satisfying any failure constraint.*

*Example 3 (Traffic Light System as SH-system).* We define the Traffic Light SH-system $TLS = (GG, C_{sys})$ by the type graph $TG$ in Fig. 1, the initial state $G_{init}$ in Fig. 2, and the following sets of rules and constraints:

- $R_{norm} = \{ArrivalCarsOne, ArrivalCarsTwo, RemoveCarsOne, RemoveCarsTwo, InfractionOn, InfractionOff\}$,
- $R_{env} = \{FailureTL, FailureCam\}$,
- $R_{rpr} = \{RepairTL, RepairCam\}$,
- $C_{consist} = \{\neg allGreen, \neg allRed\}$,
- $C_{fail} = \{TLSupFailure, CamSupFailure\}$.

The normal rule *ArrivalCarsOne* is depicted in Fig. 3 and models that one or more cars arrive at a traffic light (1:TL) while all of the crossing's lights are red. The NAC in Fig. 3 means that in this situation, no cars arrive at the other direction's traffic light (3:TL). Applying this rule, the traffic light in the direction of the arriving cars (1:TL) switches to green. Rule *ArrivalCarsTwo* (see Fig. 4) models the arrival of one or more cars at a red traffic light (2:TL) where no cars have been before, while at the same time the traffic light for the other direction (3:TL) shows green and there are already cars going in this direction. This rule causes a change of the traffic light colors in both directions. Rules *RemoveCarsOne* and *RemoveCarsTwo* are the inverse rules (with $L$ and $R$ exchanged) of the arrival rules in Fig. 3 and 4, and model the reduction of traffic at a traffic light. Rule *InfractionOn* is shown in Fig. 5 and models the situation that a car is passing the crossroad at a red light: the signal *infraction* of both the supervisor and the center of operations is set to true and the corresponding camera is starting to operate. The rule ensures that the corresponding camera is connected, using the edge attribute signal = true for edge 13:CamSup. Rule *InfractionOff* (not depicted) models the inverse action, i.e. the infraction attribute is set back to false, and the camera stops running.

The environment rules are shown in Fig. 6. They model the signal disconnection of a traffic light and a camera, respectively. The repair rules (not depicted) are defined as inverse rules of the environment rules and set the signal attributes back to true.

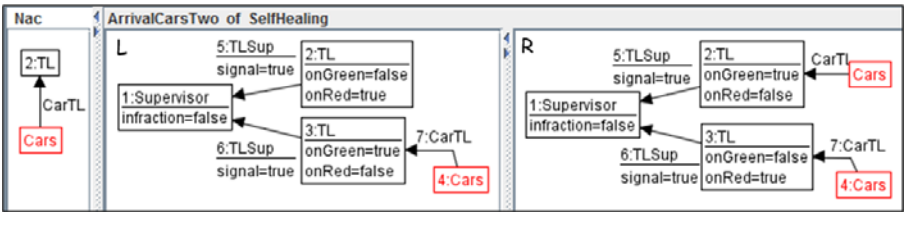



**Fig. 4.** TLS rule *ArrivalCarsTwo*

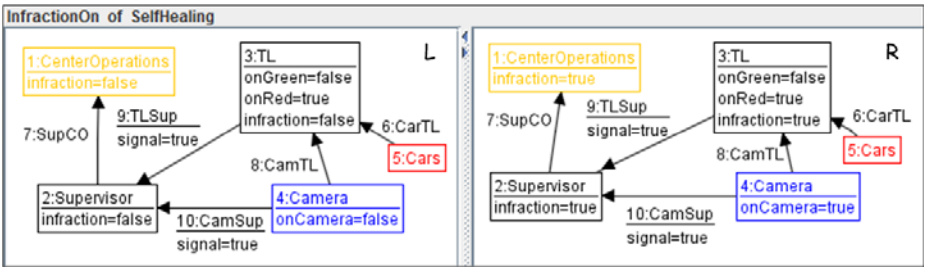



**Fig. 5.** Normal rule *InfractionOn* of *TLS*

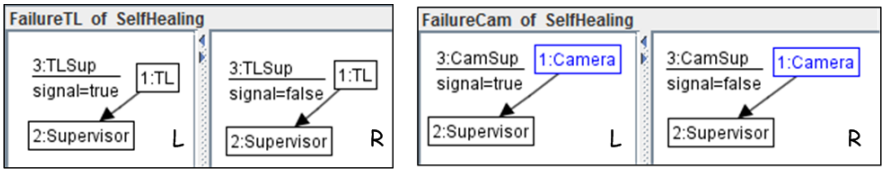



**Fig. 6.** Environment rules *FailureTL* and *FailureCam* of *TLS*

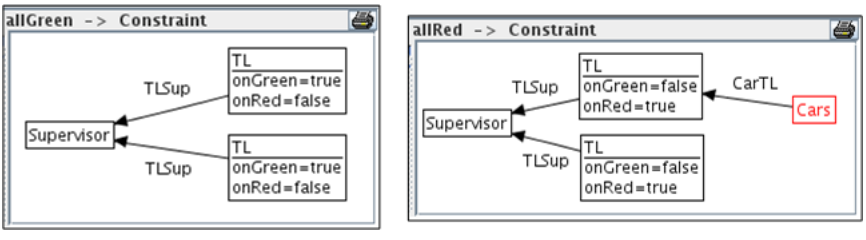



**Fig. 7.** Consistency constraint graphs of *TLS*

The consistency constraints model the desired properties that we always want to have crossroads with at least one direction showing red lights ($\neg$ *allGreen*) and avoiding all traffic lights red when there is traffic ($\neg$ *allRed*). The corresponding constraint graphs (without negation) are shown in Fig. 7. The failure constraints *TLSupFailure* and *CamSupFailure* express that either a traffic light or a camera is disconnected (the constraint graphs correspond to the right-hand sides of the environment rules in Fig. 6).

## 5 Consistency and Operational Properties of SH-Systems

In this section, we define desirable consistency and operational properties of SH-Systems. We distinguish system consistency, where all reachable states are consistent, and normal state consistency, where the initial state $G_{init}$ and all states reachable by normal rules are normal states. Environment rules, however,

may lead to failure states, which should be repaired by repair rules. We start with consistency properties:

**Definition 5 (Consistency Properties).** *An SH-System SHS is called*

1. *system consistent, if all reachable states are consistent, i.e.* $Reach(SHS) = Consist(SHS)$;
2. *normal state consistent, if the initial state is normal and all normal rules preserve normal states, i.e.*
   $G_{init} \in Norm(SHS)$ *and* $\forall G_0 \stackrel{p}{\Longrightarrow} G_1$ *via* $p \in R_{norm}$
   $[\ G_0 \in Norm(SHS) \Rightarrow G_1 \in Norm(SHS)\ ]$

*Example 4 (Consistency Properties of TLS)* The SH-System TLS is system consistent, because for all $C \in C_{consist}$ $G_{init} \models C$ and for all $G_0 \stackrel{p}{\Longrightarrow} G_1$ via $p \in R_{sys}$ and $G_0 \in Consist(SHS)$ we also have $G_1 \in Consist(SHS)$. Similarly, TLS is normal state consistent, because $G_{init} \in Norm(SHS)$ and for all $G_0 \stackrel{p}{\Longrightarrow} G_1$ via $p \in R_{norm}$ for all $C \in C_{fail}$ $[\ G_0 \not\models C \Rightarrow G_1 \not\models C\ ]$. In both cases this can be concluded by inspection of the corresponding rules, constraints and reachable states. Moreover, there are also general conditions, which ensure the preservation of graph constraints by rules, but this discussion is out of scope for this paper.

Now we consider the operational properties: one of the main ideas of SH-Systems is that they are monitored in regular time intervals by checking, whether the current system state is a failure state. In this case one or more failures have occurred in the last time interval, which are caused by failure rules, provided that we have normal state consistency. With our self-healing property below we require that each failure state can be repaired leading again to a normal state. Moreover, strongly self-healing means that the normal state after repairing is the same as if no failure and repairing would have been occurred.

**Definition 6 (Self-healing Properties).** *An SH-System SHS is called*

1. *self-healing, if each failure state can be repaired, i.e.*
   $\forall G_{init} \Rightarrow^* G$ *via* $(R_{norm} \cup R_{env})$ *with* $G \in Fail(SHS)$
   $\exists\ G \Rightarrow^+ G'$ *via* $R_{rpr}$ *with* $G' \in Norm(SHS)$
2. *strongly self-healing, if each failure state can be repaired strongly, i.e.*
   $\forall G_{init} \Rightarrow^* G$ *via* $(p_1 \dots p_n) \in (R_{norm} \cup R_{env})^*$ *with* $G \in Fail(SHS)$
   $\exists\ G \Rightarrow^+ G'$ *via* $R_{rpr}$ *with* $G' \in Norm(SHS)$ *and*
   $\exists\ G_{init} \Rightarrow^* G'$ *via* $(q_1 \dots q_m) \in R^*_{norm}$,
   *where* $(q_1 \dots q_m)$ *is subsequence of all normal rules in* $(p_1 \dots p_n)$.

*Remark 1.* By definition, each strongly self-healing SHS is also self-healing, but not vice versa. The additional requirement for strongly self-healing means, that the system state $G'$ obtained after repairing is not only normal, but can also be generated by all normal rules in the given mixed sequence $(p_1 \dots p_n)$ of normal and environment rules, as if no environment rule would have been applied. We will see that our SH-System TLS is strongly self-healing, but a modification of TLS, which counts failures, even if they are repaired later, would only be self-healing, but not strongly self-healing.

Another important property of SH-Systems is deadlock-freeness, meaning that no reachable state is a deadlock. A stronger liveness property is strong cyclicity, meaning that each pair of reachable states can be reached from each other. Note that this is stronger than cyclicity meaning that there are cycles in the reachability graph. Strong cyclicity, however, implies that each reachable state can be reached arbitrarily often. This is true for the TLS system, but may be false for other reasonable SH-Systems, which may be only deadlock-free. Moreover, we consider "normal deadlock-freeness" and "normal strong cyclicity", where we only consider normal behavior defined by normal rules.

**Definition 7 (Deadlock-Freeness and Strong Cyclicity Properties).** *An SH-System SHS is called*

1. *deadlock-free, if no reachable state is a deadlock, i.e.* $\forall G_0 \in Reach(SHS)\ \exists\ G_0 \overset{p}{\Longrightarrow} G_1$ *via* $p \in R_{sys}$
2. *normal deadlock-free, if no state reachable via normal rules is a (normal) deadlock, i.e.* $\forall G_{init} \Rightarrow^* G_0$ *via* $R_{norm}\ \exists\ G_0 \overset{p}{\Longrightarrow} G_1$ *via* $p \in R_{norm}$
3. *strongly cyclic, if each pair of reachable states can be reached from each other, i.e.* $\forall G_0, G_1 \in Reach(SHS)\ \exists\ G_0 \Rightarrow^* G_1$ *via* $R_{sys}$
4. *normally cyclic, if each pair of states reachable by normal rules can be reached from each other by normal rules, i.e.* $\forall G_{init} \Rightarrow^* G_0$ *via* $R_{norm}$ *and* $G_{init} \Rightarrow^* G_1$ *via* $R_{norm}$ *we have* $\exists\ G_0 \Rightarrow^* G_1$ *via* $R_{norm}$

*Remark 2.* If we have at least two different reachable states (rsp. reachable by normal rules), then "strongly cyclic" (rsp. "normally cyclic") implies "deadlock-free" (rsp. "normal deadlock-free"). In general properties 1 and 2 as well as 3 and 4 are independent from each other. But in Thm. 3 we will give sufficient conditions s.t. "normal deadlock-free" implies "deadlock-free" (rsp. "normally cyclic" implies "strongly cyclic" in Thm. 4).

## 6 Analysis and Verification of Operational Properties

In this section, we analyze the operational properties introduced in section 5 and give static sufficient conditions for their verification. The full proofs of our theorems are given in [9].

First, we define direct and normal healing properties, which imply the strong self-healing property under suitable conditions in Thm. 1. In a second step we give static conditions for the direct and normal healing properties in Thm. 2, which by Thm. 1 are also sufficient conditions for our self-healing properties. Of course, we have to require that for each environment rule, which may cause a failure there are one or more repair rules leading again to a state without this failure, if they are applied immediately after its occurrence. But in general, we cannot apply the repair rules directly after the failure, because other normal and environment rules may have been applied already, before the failure is monitored. For this reason we require in Thm. 1 that each pair $(p, q)$ of environment rules $p$

and normal rules $q$ is sequentially independent. By the Local Church-Rosser theorem for algebraic graph transformation [7](Thm 5.12) sequential independence of $(p, q)$ allows one to switch the corresponding direct derivations in order to prove Thm. 1. For the case with nested application conditions including NACs we refer to [31]. Moreover, the AGG tool can calculate all pairs of sequential independent rules with NACs before runtime.

**Definition 8 (Direct and Normal Healing Properties).** *An SH-System SHS has the*

1. *direct healing property, if the effect of each environment rule can be repaired directly, i.e.* $\forall G_0 \overset{p}{\Longrightarrow} G_1$ *via* $p \in R_{env}$ $\exists\ G_1 \overset{p'}{\Longrightarrow} G_0$ *via* $p' \in R_{rpr}$
2. *normal healing property, if the effect of each environment rule can be repaired up to normal transformations, i.e.* $\forall G_0 \overset{p}{\Longrightarrow} G_1$ *via* $p \in R_{env}$ $\exists\ G_1 \Rightarrow^+ G_2$ *via* $R_{rpr}$ *s.t.* $\exists\ G_0 \Rightarrow^* G_2$ *via* $R_{norm}$

*Remark 3.* The direct healing property allows one to repair each failure caused by an environment rule directly by reestablishing the old state $G_0$. This is not required for the normal healing property, but it is required only that the repaired state $G_2$ is related to the old state $G_0$ by a normal transformation. Of course, the direct healing property implies the normal one using $G_2 = G_0$.

**Theorem 1 (Analysis of Self-healing Properties).** *An SH-System SHS is*

*I. strongly self-healing, if we have properties 1, 2, and 3 below*
*II. self-healing, if we have properties 1, 2 and 4 below*

1. *SHS is normal state consistent*
2. *each pair* $(p, q) \in R_{env} \times R_{norm}$ *is sequentially independent*
3. *SHS has the direct healing property*
4. *SHS has the normal healing property*

In the following Thm. 2 we give static conditions for direct and normal healing properties. In part 1 of Thm. 2 we require that for each environment rule $p$ the inverse rule $p^{-1}$ is isomorphic to a repair rule $p'$. Two rules are isomorphic if they are componentwise isomorphic. For $p = (L \leftarrow I \rightarrow R)$ with negative application condition $nac : L \rightarrow N$ it is possible (see [7] Remark 7.21) to construct $p^{-1} = (R \leftarrow I \rightarrow L)$ with equivalent $nac' : R \rightarrow N'$. In part 2 of Thm. 2 we require as weaker condition that each environment rule p has a corresponding repair rule $p'$, which is not necessarily inverse to $p$. It is sufficient to require that we can construct a concurrent rule $p *_R p'$ which is isomorphic to a normal rule $p''$. For the construction and corresponding properties of inverse and concurrent rules, which are needed in the proof of Thm. 2 we refer to [7].

**Theorem 2 (Static Conditions for Direct/Normal Healing Properties)**

1. *An SH-System SHS has the direct healing property, if for each environment rule there is an inverse repair rule, i.e.* $\forall p \in R_{env}\ \exists\ p' \in R_{rpr}$ *with* $p' \cong p^{-1}$

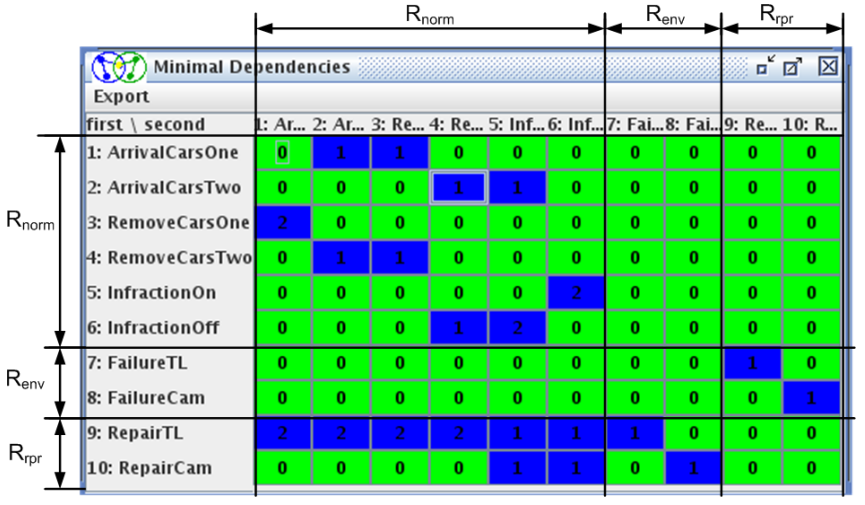



**Fig. 8.** Dependency Matrix of TLS in AGG

2. *An SH-System SHS has the normal healing property if for each environment rule there is a corresponding repair rule in the following sense:*
   $\forall p = (L \leftarrow K \rightarrow R) \in R_{env}$ *we have*
   a) *repair rule* $p' = (L' \leftarrow^{l'} K' \rightarrow^{r'} R')$ *with* $l'$ *bijective on nodes, and*
   b) *an edge-injective morphism* $e : L' \rightarrow R$ *leading to concurrent rule* $p *_R p'$*, and*
   c) *normal rule* $p'' \in R_{norm}$ *with* $p *_R p' \cong p''$

*Remark 4*. By combining Thm. 1 and Thm. 2 we obtain static conditions ensuring that an SH-System $SHS$ is strongly self-healing and self-healing, respectively.

*Example 5 (Direct Healing Property of TLS).* TLS has direct healing property because "RepairTL" rsp. "RepairCam" are inverse to "FailureTL" resp. "FailureCam" and each pair $(p, q) \in R_{env} \times R_{norm}$ is sequentially independent according to the dependency matrix of TLS in Fig. 8.

In the following Thm. 3 and Thm. 4 we give sufficient conditions for deadlock-freeness and strong cyclicity which are important liveness properties. Here we mainly use a stepwise approach. We assume to have both properties for normal rules and give additional static conditions to conclude the property for all system rules. The additional conditions are sequentially and parallel independence and a direct correspondence between environment and repair rules, which should be inverse to each other. Similar to sequential independence, also parallel independence of rules $(p, q)$ can be calculated by the AGG tool before runtime.

**Theorem 3 (Deadlock-Freeness).** *An SH-System SHS is deadlock-free, if*

1. *SHS is normally deadlock-free, and*

2. *Each pair* $(p, q) \in (R_{env} \cup R_{rpr}) \times R_{norm}$ *is sequentially and parallel independent.*

**Theorem 4 (Strong Cyclicity).** *An SH-System SHS is strongly cyclic, given*
*I. properties 1 and 2, or*
*II. properties 1, 3 and 4 below.*

1. *For each environment rule there is an inverse repair rule and vice versa.*
2. *For each normal rule there is an inverse normal rule.*
3. *SHS is normally cyclic.*
4. *Each pair* $(p, q) \in (R_{env} \cup R_{rpr}) \times R_{norm}$ *is sequentially independent.*

*Remark 5*. In part I of Thm. 4, we avoid the stepwise approach and any kind of sequential and parallel independence by the assumption that also all normal rules have inverses, which is satisfied for our TLS.

*Example 6 (Strong Cyclicity and Deadlock-Freeness of TLS)*
We use part I of Thm. 4 to show strong cyclicity. Property 1 is satisfied because "FailureTL" and "RepairTL" as well as "FailureCam" and "RepairCam" are inverse to each other. Property 2 is satisfied because "ArrivalCarsOne(Two)" and "RemoveCarsOne(Two)" as well as "InfractionOn" and "InfractionOff" are inverse to each other. Moreover, deadlock-freeness of TLS follows from strong cyclicity by remark 2. Note that we cannot use part II of Thm. 4 for our example TLS, because e.g. ("RepairTL", "ArrivalCarsOne") is not sequentially independent.

## 7 Conclusion

In this paper, we have modeled and analyzed self-healing systems using algebraic graph transformation and graph constraints. We have distinguished between consistency properties, including system consistency and normal state consistency,

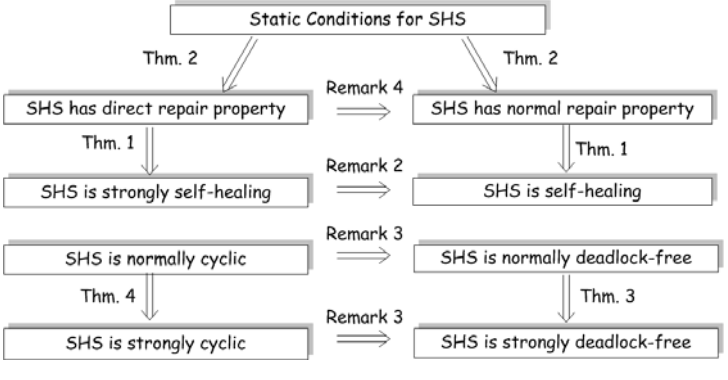



**Fig. 9.** Operational properties of self-healing systems

and operational properties, including self-healing, strongly self-healing, deadlock-freeness, and strong cyclicity. The main results concerning operational properties are summarized in Fig. 9, where most of the static conditions in Thms. 1- 4 can be automatically checked by the AGG tool.

All properties are verified for our traffic light system. Note that in this paper, the consistency properties are checked by inspection of corresponding rules, while the operational properties are verified using our main results. Work is in progress to evaluate the usability of our approach by applying it to larger case studies. As future work, we will provide analysis and verification of consistency properties using the theory of graph constraints and nested application conditions in [31]. Moreover, we will investigate how far the techniques in this paper for SH-systems can be used and extended for more general self-adaptive systems.

## References

1. 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. In: Software Engineering for Self-Adaptive Systems, pp. 48–70 (2009)
2. Andersson, J., Lemos, R., Malek, S., Weyns, D.: Modeling dimensions of self-adaptive software systems. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525, pp. 27–47. Springer, Heidelberg (2009)
3. Rodosek, G.D., Geihs, K., Schmeck, H., Burkhard, S.: Self-healing systems: Foundations and challenges. In: Self-Healing and Self-Adaptive Systems, Germany. Dagstuhl Seminar Proceedings, vol. 09201 (2009)
4. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer 36(1), 41–50 (2003)
5. 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)
6. Bucchiarone, A., Pelliccione, P., Vattani, C., Runge, O.: Self-repairing systems modeling and verification using AGG. In: WICSA 2009 (2009)
7. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS Monographs in Theor. Comp. Science. Springer, Heidelberg (2006)
8. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G. (eds.): Handbook of Graph Grammars and Computing by Graph Transformation. Applications, Languages and Tools, vol. 2. World Scientific, Singapore (1999)
9. Ehrig, H., Ermel, C., Runge, O., Bucchiarone, A., Pelliccione, P.: Formal analysis and verication of self-healing systems: Long version. Technical report, TU Berlin (2010), `http://www.eecs.tu-berlin.de/menue/forschung/forschungsberichte/2010`
10. Perry, D., Wolf, A.: Foundations for the Study of Software Architecture. SIGSOFT Softw. Eng. Notes 17(4), 40–52 (1992)
11. Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. In: FOSE, pp. 259–268 (2007)
12. Floch, J., Hallsteinsen, S., Stav, E., Eliassen, F., Lund, K., Gjorven, E.: Using architecture models for runtime adaptability. IEEE Software 23(2), 62–70 (2006)

13. Garlan, D., Schmerl, B.: Model-based adaptation for self-healing systems. In: WOSS 2002, pp. 27–32. ACM, New York (2002)
14. 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 2008), pp. 508–516. ACM Press, New York (2008)
15. Bucchiarone, A.: Dynamic software architectures for global computing systems. PhD thesis, IMT Institute for Advanced Studies, Lucca, Italy (2008)
16. 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, New York (2006)
17. Baresi, L., Heckel, R., Thone, S., Varro, D.: Style-based refinement of dynamic software architectures. In: WICSA 2004. IEEE Computer Society, Los Alamitos (2004)
18. Hirsch, D., Inverardi, P., Montanari, U.: Modeling software architectures and styles with graph grammars and constraint solving. In: WICSA, pp. 127–144 (1999)
19. Métayer, D.L.: Describing software architecture styles using graph grammars. IEEE Trans. Software Eng. 24(7), 521–533 (1998)
20. Kastenberg, H., Rensink, A.: Model checking dynamic states in groove. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 299–305. Springer, Heidelberg (2006)
21. Aguirre, N., Maibaum, T.S.E.: Hierarchical temporal specifications of dynamically reconfigurable component based systems. ENTCS 108, 69–81 (2004)
22. Rensink, A., Schmidt, A., Varr'o, D.: Model checking graph transformations: A comparison of two approaches. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 226–241. Springer, Heidelberg (2004)
23. Reisig, W.: Petri Nets: An Introduction. EATCS Monographs on Theoretical Computer Science, vol. 4. Springer, Heidelberg (1985)
24. Spanoudakis, G., Zisman, A., Kozlenkov, A.: A service discovery framework for service centric systems. In: IEEE SCC, pp. 251–259 (2005)
25. Canfora, G., Penta, M.D., Esposito, R., Villani, M.L.: An approach for qos-aware service composition based on genetic algorithms. In: GECCO, pp. 1069–1075 (2005)
26. Zeng, L., Benatallah, B., Dumas, M., Kalagnanam, J., Sheng, Q.Z.: Quality driven web services composition. In: WWW, pp. 411–421 (2003)
27. Baresi, L., Guinea, S., Pasquale, L.: Self-healing BPEL processes with Dynamo and the JBoss rule engine. In: ESSPE 2007, pp. 11–20. ACM, New York (2007)
28. Colombo, M., Nitto, E.D., Mauri, M.: Scene: A service composition execution environment supporting dynamic changes disciplined through rules. In: ICSOC, pp. 191–202 (2006)
29. Rukzio, E., Siorpaes, S., Falke, O., Hussmann, H.: Policy based adaptive services for mobile commerce. In: WMCS 2005. IEEE Computer Society, Los Alamitos (2005)
30. Inverardi, P., Pelliccione, P., Tivoli, M.: Towards an assume-guarantee theory for adaptable systems. In: SEAMS, pp. 106–115. IEEE Computer Society, Los Alamitos (2009)
31. Ehrig, H., Habel, A., Lambers, L.: Parallelism and Concurrency Theorems for Rules with Nested Application Conditions. In: EC-EASST (to appear, 2010)