Self-Repairing Systems Modeling and Verification using AGG

Antonio Bucchiarone
FBK-IRST, Trento, Italy
via Sommarive 18, 38050
Trento, Italy
bucchiarone@fbk.eu

Patrizio Pelliccione, Charlie Vattani
Dipartimento di Informatica
Università dell'Aquila
L'Aquila, Italy
patrizio.pelliccione@di.univaq.it

Olga Runge
Department of Software Engineering and
Theoretical Computer Science
Technical University of Berlin
olga@cs.tu-berlin.de

***Abstract*—Self-Repairing (or healing) systems are systems equipped with a mechanism that monitors the system behaviour to determine whether it behaves within prefixed parameters. If a deviation exists, then the system itself is in charge of adapting its configuration. In this paper we show how to model self-repairing systems by means of Dynamic Software Architectures (DSAs). DSAs are formalized as Typed (hyper) Graph Grammars (TGGs) and this formalization enables verification of correctness and completeness of self-repairing systems. DSAs are modeled and verified by using the Attributed Graph Grammar system (AGG). The overall approach is applied to a traffic light system case study.**

***Keywords*-Self-repairing systems, dynamic software architectures, healing systems, graph grammars, AGG.**

I. Introduction

Traditional systems are usually designed with a set of requirements and a precise context that provides all the resources necessary to run system services in mind. On the contrary, the intended lifespan of modern software systems makes requirements and context evolution the norm rather than the exception. When the requirements and the context evolve, a software system must be able to face changes in an effective way with minimal human intervention otherwise it will soon become obsolete. Software systems must be designed with run-time evolution in mind and, at the same time, solutions must be adopted to keep the (user's perceived as well as system intrinsic) dependability at a satisfactory level [1]. Mechanisms for run-time adaptation are needed to provide good reactions to (system and context) changes.

Dynamic Software Architecture (DSA) [2], [3], [4], [5] plays a central role in enabling system evolution: it has to support models at run-time, continuous verification and self-repairing by providing a suitable infrastructure. When the changes are initiated and accessed internally, DSAs keep the name of *self-repairing* DSAs [6]. Different approaches for self-repairing DSA have been proposed in literature, e.g., [6], [7]. Typically, they are bound to particular languages and models. In this paper we are aimed at understanding the main notions of DSAs by abstracting away from particular languages and notions. In other words, we want to give a uniform and formal representation of DSAs that is abstract enough to cover most of the features presented in previous works. In this sense, our work is in line with other researches (e.g., [4], [8], [9]). The formalization of self-repairing DSAs that we provide in this paper, obtained by extending the formalization already presented in [10], enables the verification of correctness and completeness of self-repairing specification. More precisely, this allows us to verify that each desirable configuration can be reached (completeness) and that for each reachable configuration that is not desirable there exist repairing productions (correctness).

In this paper we use AGG[1] as the framework to model and verify DSAs. The use of AGG solves many limitations emerged by using Maude and Alloy [11]. In [10] we use hypergraph grammars as a formal framework for mapping the different notions of dynamism. In [11] we presented two ways to model and analyze DSAs: (i) by using ordinary typed hypergraph transformation techniques implemented in Alloy [12], and (ii) by means of a process algebraic presentation of graph transformation implemented in Maude [13]. With the use of AGG we are able to verify completeness and correctness of self-repairing systems by means of a unique formalism (i.e., T-typed Hypergraph Grammars) and a unique modeling and verification tool.

The paper is organized as following: Sect. II sets the context of the paper and relates this paper with the state of the art in dynamic software architectures. Section III shows how to represent self-repairing software architectures as T-typed hypergraph grammars. Section IV presents the AGG framework that we used to model and verify self-repairing systems. We apply the methodology to an existing automated traffic light system: Sect. V describes the use of AGG for modeling the case study, while Sect. VI describes the use of AGG for verifying the system. Finally, Sect. VII concludes the paper and discusses future works.

II. Dynamic Software Architectures

In the last years several research papers and projects have had their dynamic system modeling and adaptation as main topic; furthermore they have provided new paradigms that extend the classic software architecture notation. For example Morrison et al in [14] define an *Active Architecture*

[1]AGG: http://tfs.cs.tu-berlin.de/agg.

as: "*A software architecture that can evolve during execution by creating, deleting, reconfiguring and moving components, connectors and their configurations at runtime*". Chatley et al. in [15] define *Plugin architectures* where components (i.e., Plugins) can be added (or deleted) to an existing system at runtime to extend (or reduce) its functionality, by preserving desired properties. Bradbury et al. in [8] define a *Self-managing architecture* as: "an architecture that not only implements the change internally but also initiates, selects, and assesses the change itself without the assistance of an external user". Finally, Hirsch et al. in [16] propose the notion of *Mode* as a new element of architectural descriptions with the goal of providing flexible support for the description and verification of complex adaptable service oriented systems. Moreover, a variety of definitions of dynamism for software architecture have been proposed in Literature. Below we list some of the most prominent definitions to show the variability of connotations that the word dynamic acquires:

Programmed [17]: all admissible changes are defined prior to runtime and are triggered by the system itself;

Self-Reparing [6]: changes are initiated and assessed internally, i.e., the runtime behavior of the system is monitored to determine whether a change is needed. In such case, a reconfiguration is automatically performed;

Ad-hoc [17]: changes are initiated by the user as part of a software maintenance task, they are defined at run-time and are not known at design-time;

Constructible [3]: it is a kind of ad-hoc mechanism but all architectural changes must be described in a given modification language, whose primitives constrain the admissible changes.

In this paper we focus on self-repairing dynamic software architectures. Researchers have differing views on what comprises research on self-repairing systems. In [18] the author proposes a taxonomy for describing the problem space for these systems and concludes the paper saying that :"*Relevant aspects of self-repairing system approaches include fault models, system responses, system completeness, and design context*". Moreover, he specifies that the SA of a self-repairing system must provide adaptation mechanisms to make the system "open" so that third-party components can be added during or after system deployment. Cheng et al. in [7] introduce a SA-based self-adaptation framework, called Rainbow, which uses external mechanisms and a SA model to monitor a managed system, detect problems, determine a course of action, and carry out the adaptation actions. In [19] some authors of the Rainbow framework extend it proposing a new language of adaptation able to capture the basic ontology, for an adaptation language that holds the promise of automating human tasks in system management. Other works have the objective to use architectural styles to support architecture-based self-adaptation. Baresi et al. [4] formalize architectural style as a typed graph transformation system. It consists of a type graph to define architectural elements and their relationships, a set of constraints to further restrict the valid models, and a set of graph transformation rules. Nodes of the type graph define the architectural elements (i.e., components, connectors, ports, interfaces, etc.). Edges define the possible relationships among these elements. Moreover, a concrete architecture is an instance graph of the type graph. Reconfiguration mechanisms are modeled using transformation rules that can be applied to change the SA configuration.

Motivation of the work

In this paper we are aimed at proposing a uniform formal specification of self-repairing DSAs that is able to cover most of the features of the languages and models proposed in Literature. We aim to build a *formal* specification in order to be able to perform the verification of correctness and completeness of self-repairing system specifications. The use of AGG is strategic since by means of this tool we are able to use a unique formalism for both modeling and verification.

III. Representing Self-Repairing SAs as T-typed hypergraph grammars

In this section, building on the formalization presented in [10], we show how to represent self-repairing software architectures as T-typed hypergraph grammars. The first definition that we introduce is the definition of *hypergraph*.

Definition 1 (Hypergraph): A *hypergraph* is a triple $H{=}(N_H,E_H,\phi_H)$, where:

- N_H is the set of nodes;
- E_H is the set of hyperedges. A hyperedge $e{\in}E_H$, $e{=}(l,\{t_1,\cdots,t_n\})$, is a pair composed of l, that is the label of the hyperedge, and $\{t_1,\cdots,t_n\}$, that is the set of tentacles of e;
- $\phi_H : E_H \rightarrow N_H^+ \times \cdots \times N_H^+$ describes the connections of the graph, where N_H^+ stands for the non-empty set of elements of N_H.

Given a hypergraph $H{=}(N_H,E_H,\phi_H)$, let $e{\in}E_H$ be a hyperedge and t a tentacle of e, we denote with $\phi_H(e){\rightarrow}\{\{n_1^{t_1},\cdots,n_k^{t_1}\},\ldots,\{n_1^{t_n},\cdots,n_h^{t_n}\}\}$ the ordered set of sets of nodes that are matched by the hyperedge e by means of the tentacles $t_1,\ldots,t_n$, respectively. We denote with $\phi_H(e)[t_i]$ the set of nodes $\{n_1^{t_i},\cdots,n_r^{t_i}\}$ that are matched by the hyperedge e by means of the tentacle t_i.

In our formalization components and connectors are represented by hyperedges. A hyperedge is composed of the component and connector name and of a set of tentacles that represent the ports of components and connectors. A node of a hypergraph represents the match between a port of a component (or connector) and a port of a different component (or connector). These matches are realized by means of the connection function ϕ_H that associates each

component and connector to an ordered and non empty sequence of matches.

In our context a hypergraph is *well-formed* iff for each tentacle t of each hyperedge e, there exists a function $\phi_H(e)[t] \rightarrow \{n_1, \cdots, n_m\}$ for some $n_1, \cdots, n_m$.

Architectural styles are defined by means of particular hypergraphs, in which nodes have also an arity that constraints the number of times a node can be matched by the connection function. The purpose of the architectural style is to describe the types of connectors and components, and to define the allowed connections among these elements. A configuration compliant to such style is described by the notion of *Typed Hypergraph*, defined in Def. 2.

Definition 2 (Typed Hypergraph): Given a hypergraph T (called the *style*), a *T-typed hypergraph* or *configuration* is a pair $\langle G, \tau \rangle$, where G is the *underlying* graph and τ is a function that maps each element of G to its element type in the style.

A T-typed hypergraph G is *well-formed* if its connection function respects the constraints imposed by the arity of each node (defined into the typed hypergraph T). With $\mathcal{H}_T$ we denote a set of T-typed hypergraphs.

The reconfiguration of a software architecture is described by a set of rewriting productions. A production $pr: \mathcal{H}_T \rightarrow \mathcal{H}_T$ is a partial, injective morphism of T-typed hypergraphs that transforms the left-hand side of the production into the right-hand side. Then, a *dynamic software architecture* is described by a T-typed hypergraph grammar (see Def. 3).

Definition 3 (T-typed) hypergraph grammar): A *(T-typed) hypergraph grammar* $\mathcal{G}$ is a tuple $\langle T, G_{in}, PR \rangle$, where T is the style, G_{in} is the *initial (T-typed) hypergraph* and PR is a set of *productions*.

Let $\mathcal{G}=\langle T, G_{in}, PR \rangle$ be a (T-typed) hypergraph grammar, and G and H two (T-typed) hypergraphs. With $G \Rightarrow_{pr} H$ we denote that G is rewritten in one step to H by using the production pr. We abbreviate the reduction sequence $G_0 \Rightarrow_{pr_0} G_1 \Rightarrow_{pr_1} G_2 \Rightarrow_{pr_2} \ldots \Rightarrow_{pr_n} G_{n+1}$ with $G_0 \Rightarrow_{pr_0;pr_1,\ldots;pr_n} G_{n+1}$. Finally, with $G \Rightarrow^+ G'$ we denote that there exists a non-empty sequence $s \in PR^+$ of derivation steps such that $G \Rightarrow_s G'$.

Moreover, given a grammar $\mathcal{G}=\langle T, G_{in}, PR \rangle$, we use the following notions:

- with $\mathcal{R}(\mathcal{G})$ we denote the set of reachable configurations, i.e., all configurations to which the initial configuration G_{in} can evolve. Formally, $\mathcal{R}(\mathcal{G})=\{G \mid G$ is a T-typed hypergraph $\wedge\ G_{in} \Rightarrow^+ G\}$;
- with $\mathcal{D}_{\mathsf{P}}(\mathcal{G})$ we denote desirable configurations, i.e., the set of all (T-typed) hypergraphs that satisfy a desired property P. Formally, $\mathcal{D}_{\mathsf{P}}(\mathcal{G})=\{G \mid G$ is a T-typed hypergraph $\wedge$ P holds in $G\}$. It is not important for the aim of this paper to better investigate the nature of the properties P. In future work we could investigate the use of graphical formalism for expressing properties, such as [20].

A. Repairing (or healing) dynamism

Repairing systems are equipped with a mechanism that monitors the system behavior to determine whether it behaves within prefixed parameters. If a deviation exists, then the system is in charge of adapting the configuration [6]. A dynamic repairing software architecture can be represented as a T-typed hypergraph grammar $\mathcal{G}_{\mathcal{R}}=\langle T, G_{in}, PR \rangle$ in which the set of productions is divided into three different subsets: $PR=PR_{pgm} \cup PR_{env} \cup PR_{rpr}$. PR_{pgm} contains productions that describe the normal and ideal behavior of the architecture, i.e., $\mathcal{G}'_{\mathcal{R}}=\langle T, G_{in}, PR_{pgm} \rangle$ is a programmed DSA (already formalized in [10] and [21]). PR_{env} contains productions that model the *environment* or, in other words, the ways in which the behavior of the architecture may deviate from the expected one (e.g., loss of communication messages, communication managed by a non-authorized connector, etc.). Productions in PR_{rpr} indicate the way in which an undesirable configuration can be repaired in order to become a valid one. That is, the left-hand side of any rule in PR_{rpr} identifies a composition pattern in the system that is undesirable.

1) Repairing dynamism - Verification aspects: Repairing dynamism specifications can be checked for correctness and completeness. Let $\mathcal{G}_{\mathcal{R}}=\langle T, G_{in}, PR \rangle$ be a T-typed hypergraph grammar, then:

- the specification is *complete* if each desirable configuration different from G_{in} can be reached by applying repairing mechanisms. Formally, for each $G \in \mathcal{D}_{\mathsf{P}}(\mathcal{G}_{\mathcal{R}})$, with $G \neq G_{in}$, $G \in \mathcal{R}(\mathcal{G}_{\mathcal{R}})$.
- the specification is *correct* if there exist repairing productions for each reachable configuration that does not belong to the set of the desirable configurations. Formally, for each $G \in \mathcal{R}(\mathcal{G}_{\mathcal{R}})$, if $G \notin \mathcal{D}_{\mathsf{P}}(\mathcal{G}_{\mathcal{R}})$ then $\exists q \in PR_{rpr}$ such that $G \Rightarrow_q G'$, for some $G' \in \mathcal{D}_{\mathsf{P}}(\mathcal{G}_{\mathcal{R}})$.

B. Self dynamism

Usually, some kind of dynamism (like programmed and repairing) is also qualified as "self", meaning that the changes are initiated by the system itself. In [8] a classification of dynamism of DSAs has been proposed. In addition to **external** reconfiguration, in which the reconfiguration rule is selected by an external source, there are two categories of dynamism:

Autonomous: the system selects one transformation among the applicable transformations in a non-deterministic way. This corresponds to the notion of internal choices in process calculi. Accordingly, we may represent such reductions by hiding the actual name of the applied rule. That is, $G \Rightarrow_{pr} G'$, in which pr is autonomous, can be represented as $G \Rightarrow_{\tau} G'$, where τ stands for a hidden change.

Pre-defined: pre-defined selection is a special case of autonomous choice, in which the system selects in a pre-defined way the appropriate transformation to apply from the set of available ones. In this case, the choice is completely deterministic (like a conditional choice *if - then - else* of process calculi). This can be mapped into hypergraph grammars as the definition of priorities in the selection of productions to be applied. As shown in [22], application conditions can be used as priorities for restricting the order in which rules are applied.

Let $\mathcal{G}_{\mathcal{S}}=\langle T,G_{in},PR_{ext}\cup PR_{self}\rangle$ be a grammar, where PR_{ext} stands for the set of all reconfigurations that are controlled by the environment, while PR_{self} contains all the autonomous productions. We say $\mathcal{G}_{\mathcal{S}}$ has (i) self dynamism if $PR_{ext}=\emptyset$, (ii) external dynamism if $P_{self}=\emptyset$, or (iii) mixed dynamism otherwise. Assuming that all rewriting steps $G\Rightarrow_{pr}G'$ are written $G\Rightarrow_{\tau}G'$ when $pr\in PR_{self}$, we define the following sets associated to the grammar $\mathcal{G}_{\mathcal{S}}$:

• the set $\mathcal{S}(\mathcal{G}_{\mathcal{S}})$ of autonomous or self reconfigurations, i.e., the set of all configurations reachable by applying autonomous changes is: $\mathcal{S}(\mathcal{G}_{\mathcal{S}}) = \{G \mid G_{in} \Rightarrow_{\tau^*} G\}$;

• the set $\mathcal{E}_c(\mathcal{G}_{\mathcal{S}})$ of reconfigurations associated to an external sequence $c=p_1 \ldots p_n$ of commands:
$\mathcal{E}_c(\mathcal{G}_{\mathcal{S}})=\{G|G_{in}\Rightarrow_c G,$ with $c=\tau^*,p_1,\tau^*,\ldots,\tau^*,p_n,\tau^*\}$. In other words, $\mathcal{E}_c(\mathcal{G}_{\mathcal{S}})$ contains all the configurations reachable from the initial configuration by applying the sequence c of external chosen rules interleaved with the application of zero or more autonomous reconfigurations.

1) Self dynamism - Verification aspects: Clearly, $\mathcal{S}(\mathcal{G}_{\mathcal{S}})$ and $\mathcal{E}_c(\mathcal{G}_{\mathcal{S}})$ are subsets of $\mathcal{R}(\mathcal{G}_{\mathcal{S}})$. Hence, we can proceed as in Sect. III-A1, and we can formulate some verification issues. The completeness and correctness properties can be specialized as follows:

• the specification is *complete - self reconfigurations* if each desirable configuration different from G_{in} can be reached by applying autonomous repairing mechanisms. Formally, for each $G\in\mathcal{D}_{\mathsf{P}}(\mathcal{G}_{\mathcal{R}})$, with $G\neq G_{in}$, $G\in\mathcal{S}(\mathcal{G}_{\mathcal{R}})$.

• the specification is *complete - external reconfigurations* if each desirable configuration different from G_{in} can be reached by applying a sequence of external chosen rules interleaved with the application of zero or more autonomous reconfigurations rules. Formally, for each $G\in\mathcal{D}_{\mathsf{P}}(\mathcal{G}_{\mathcal{R}})$, with $G\neq G_{in}$, $G\in\mathcal{E}_c(\mathcal{G}_{\mathcal{R}})$.

• the specification is *correct - self reconfigurations* if there exist self repairing productions for each configuration that does not belong to the set of the desirable configurations. Formally, for each $G\in\mathcal{S}(\mathcal{G}_{\mathcal{R}})$, if $G\notin\mathcal{D}_{\mathsf{P}}(\mathcal{G}_{\mathcal{R}})$ then $\exists q\in PR_{self}$ such that $G\Rightarrow_q G'$, for some $G'\in\mathcal{D}_{\mathsf{P}}(\mathcal{G}_{\mathcal{R}})$.

• the specification is *correct - external reconfigurations* if there exist external repairing productions for each configuration that does not belong to the set of the desirable configurations. Formally, for each $G\in\mathcal{E}_c(\mathcal{G}_{\mathcal{R}})$, if $G\notin\mathcal{D}_{\mathsf{P}}(\mathcal{G}_{\mathcal{R}})$ then $\exists q\in PR_{ext}$ such that $G\Rightarrow_q G'$, for some $G'\in\mathcal{D}_{\mathsf{P}}(\mathcal{G}_{\mathcal{R}})$.

IV. ATTRIBUTED GRAPH GRAMMAR (AGG) SYSTEM

AGG is a well established graph transformation environment developed at TU Berlin. The AGG environment is designed as a tool for editing directed, typed and attributed graphs in order to define a graph grammar. This graph grammar is the input of the graph transformation engine of AGG that performs transformation steps for user-selected productions. The AGG language is a rule based visual language supporting an algebraic approach to graph transformation. It allows specifying and rapid prototyping applications with complex, graph structured data.

The application's behavior is described by graph rules using an *if-then* programming style. The application of a graph rule changes the graph structure. AGG supports conditional graph transformations. Transformation rules may specify positive and negative application conditions. A Positive Application Condition (PAC) indicates the presence of a given graph structure (i.e., a certain combination of nodes and edges). A Negative Application Condition (NAC) indicates the required absence of a graph structure. Additionally, rules may have attribute conditions being *Boolean* Java expressions. The control flow is implicitly handled by the dependency of rule applications. Alternatively, rule applications may be controlled by graph constraints and explicit control constructs such as layered graph transformation.

The user interface of AGG for editing graph grammars consists of graphical editors for graphs and rules as well as a textual editor for Java expressions integrated into the visual editors. Moreover, visual interpretation and debugging are supported. The strength of AGG lies in its several kinds of validations. The validation tool offers several features, namely: (i) graph parse, (ii) consistency checking of graphs, (iii) termination criteria for controlled rule applications, (iv) critical pair analysis for conflict detection of graph rules in concurrent transformations, and (v) applicability and non-applicability check for rule sequences.

In the following we focus on *consistency checking of graphs* and *critical pair analysis*, which are interesting for our perspectives. Section VI will put in practice AGG by showing how this tool can be used for checking correctness and completeness of self-repairing system specifications.

A. Consistency Check of Graphs

AGG provides consistency control mechanisms, which are able to check if a given graph satisfies certain consistency conditions (constraints) specified for a graph grammar. These consistency conditions are Boolean formulas of atomic graph constraints which describe basic properties of graphs as e.g., the existence of certain elements, independent of a particular rule. AGG allows us to convert global consistency conditions into post application conditions for individual rules [23]. A so-extended rule is applicable to a consistent graph if and only if the derived graph is consistent. A graph grammar is consistent if the start graph

satisfies the consistency conditions and the rules preserve this property. Since a consistency condition is a Boolean formula the negation of an atomic graph constraint inside it will express the absence of some graph structure in a graph G. We will exploit this feature of consistency conditions in our case study. There are two ways how to use consistency conditions for checking consistency of graphs during graph transformation. The first way is to check consistency conditions on a graph globally. In other words, they will be checked after each rule application during graph transformation. If graph consistency is satisfied, the transformation step is accepted, otherwise it is refused and another applicable rule is taken for the next step. The second way is to convert global consistency conditions to the Post Application Conditions (PACs) for individual rule or for all rules. This converting will not be done automatically. The user has to choose which consistency conditions for which rule(s) should be converted to PACs. Then, they are used for checking graph consistency after the rule is applied. Transformation step will be refused when consistency check failed.

In our case study we do not use converting consistency conditions into Post Application Conditions for individual rules but we let to check consistency conditions globally after each rule application during graph transformation.

B. Critical Pair Analysis

Critical pair analysis (CPA) [24], [25] is the static analysis of potential conflicts and dependencies of transformation rules based on the notion of independence of graph transformations i.e., when two transformations are neither causally dependent nor in conflict.

If two transformations are mutually independent, they can be applied in any order yielding the same result. This is a case of parallel independence. In general, there are three reasons why rule applications can be conflicting: (i) one rule application deletes a graph object which is in the match of another rule application; (ii) one rule application generates graph objects in a way that a graph structure would occur which is prohibited by a NAC of another rule application; (iii) one rule application changes attributes being in the match of another rule application.

Because of the duality between conflicts and dependencies, critical pair analysis can be used to find all potential dependencies of rules. From the set of all critical pairs we can extract the nodes and edges which cause conflicts or dependencies.

Surveying the results of the analysis, the designer has to decide which dependencies or conflicts really represent errors. Not every conflict represents an error. If two rules are meant to be applied alternatively, the conflict simply reflects this requirement at the object level. Concerning dependencies, improvements of the model may be proposed whenever the dependencies defer from the control flow.

An absence of conflict and dependency could either indicate that rules may be applied concurrently or that the specification could be enhanced by explicitly modeling the restrictions that lead to a rather sequential execution. Then, the results of CPA can give valuable hints for improving the model both from the presence and the absence of conflicts and dependencies.

The AGG tool provides a graphical user interface for generating and browsing through computed critical pairs. It gives clear and sufficient information about critical pairs and allows us to consider a critical pair in more detail, i.e., for each pair, the two rules and all critical overlapping graphs with corresponding matches are shown.

In general, CPA is time and space expensive. However, the number of critical pairs can be reduced drastically by the use of multiplicity constraints. Moreover, the set of critical pairs can be reduced considerably by defining graph constraints. Not all kinds of graph constraints can be successfully used for reduction, because CPA operates: i) on minimal graphs of critical situation, and ii) in variable attribute context where not all attribute values and conditions can be evaluated. AGG allows optional use of graph consistency constraints for CPA. Sometimes it makes sense to abstain from checking consistency constraints during critical pair analysis especially when graph constraints aim to check not only graph structure but mainly the attribute values of the components.

V. TRAFFIC LIGHT SYSTEM CASE STUDY

In this section we apply the methodology presented in this paper to an existing automated traffic light system[2]. The new traffic light technology is based upon electromagnetic spires buried some centimeters. Also, it is underneath the asphalt of the most congested paths. The spires register traffic data and send them to others components of the system, which will handle the information. The system helps the infraction system by making it incontestable. In fact, the Traffic Light System (TLS) is connected to cameras which record videos of the violations and automatically send them to the center of operations. The TLS is responsible for regulating the traffic lights in a "smart" way. In particular, it provides the following functionalities:

Switching the traffic light signal: the spires send the electricity created by the passing of the vehicles to an Electronic Control Unit (ECU), which registers the medium speed of the cars, hence the intensity of traffic. The `ECU` component sends the data to another ECU component that is called `Supervisor`. The `Supervisor` imparts a cadence to the green and the red lights of several traffic lights in the same street. The system gives a priority if more cars are in line in the same direction.

Management of infractions: each camera is connected to the `Supervisor`, which constantly controls the traffic light

[2]http://www.lasemaforica.com/vista_red.htm

signal and the ECU. If a vehicle transits upon the spires while the street light is red, the Supervisor triggers the camera, which starts recording. When the light is green, it ignores the spires and does not trigger the cameras. Contextually with the infraction event, the camera sends the records to the video recorder of the center of operations, which stores date, time and Supervisor ID in order to avoid legal challenges.
Error management: each time there is an error in the system the supervisor detects it and sends the error type to the CenterOperations. Errors can be the breaking down of the components as well as the loss of connection from one of the components of the TLS with the rest of the system. The CenterOperations triggers a repairing procedure to restore system faults. In this scenario the dynamism is given by the traffic flow. Vehicles join and leave the system continuously, and there is no way to predict it, and so we cannot predict the traffic light behavior. In the following we show the TLS configurations:

- *Traffic flow:* as the cars get to the cross road, the traffic flow is stored in the Spire component and sent to the ECU component. Then, the ECU component forwards the information to the Supervisor component which will manage it;
- *Switching:* switching between green/red light. This is handled by the Supervisor component;
- *Light Check:* checks that each cross road has only one green light turned on. This checking is carried on by the Supervisor component;
- *Broken Camera:* the Supervisor component checks if there is an error signal linked to the Camera component and sends it to the CenterOperations component;
- *Broken traffic light:* if there is a loss of a traffic light signal, the system handles it in order to repair it, if possible (obviously some faults are unrecoverable);
- *No traffic flow:* when there is no traffic flow then every red traffic light must be turned on, and every green traffic light must be turned off.

A. TLS Components and Style

This section introduces the components and the style of the TLS. The architecture of the system is composed by five components that are shown in Fig. 1.

As described in Sect. III, each component and connector is represented as a hyperedge where its tentacles represent ports. For example, the Supervisor component of Fig. 1 has three ports: $supCO$, $supCam$ and $supTL$ to connect it to the components CenterOperations, Camera and TL, respectively. As we can see from the typed graph (or style) of Fig. 2, the TLS system is composed of only one CenterOperations component, of one Supervisor, of two TL components, two Camera components, and two optional Car components, one for each TL.

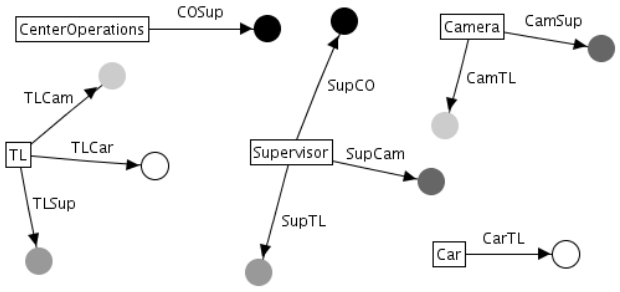

Figure 1. Basic Components of TLS

It is important to model how many objects can be connected through an instance of a hyperedge type. AGG allows the specification of the multiplicity of the hyperedges of the typed graphs. Expressing a multiplicity at the target end of a hyperedge type means specifying the number of nodes which may be connected to the source node across hyperedges of the given hyperedge type. A multiplicity at the source end of a hyperedge type is interpreted similarly. For instance, by referring to Fig. 2, the $SupTL$ tentacle (port) is attached with the component Supervisor at its source, and with two connections $TL{-}Supervisor$ at its target. Each $TLSup$ hyperedge is attached with the component TL at its source, and with the connection $TL{-}Supervisor$ at its target. Setting multiplicities in AGG has several positive effects since they pose additional constraints on a hypergraph:
1) the number of NACs for rules might decrease;
2) the number of consistency conditions might decrease;
3) the efficiency of the critical pair analysis might increases.

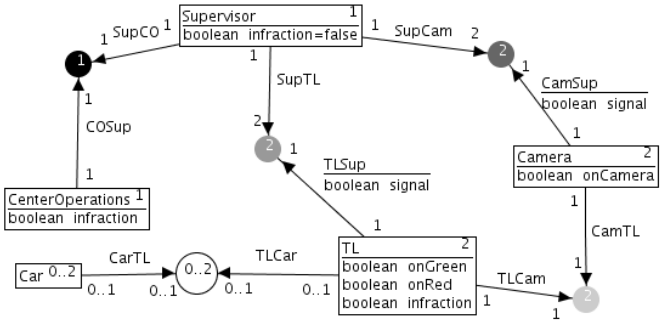

Figure 2. Style of TLS

Points 1 and 2 are important since those are the instruments that we use to check correctness and completeness of our grammar and then decreasing the number of NACs and consistency conditions means decreasing the amount of work to the programmer. As explained in Sect. IV, AGG enables attributing nodes and tentacles. This is very useful especially for expressing consistency conditions of the grammar. In the TLS, the component TL has three Boolean attributes: $onGreen$, $onRed$ and $infraction$. They express whether a green/red light is on/off and whether there is an infraction

going on. The ports `TLSup` and `CamSup` have also the *signal* Boolean attribute. This is used to model whether or not there is a communication between `TL` and `Supervisor` or between `Camera` and `Supervisor`. Figure 3 shows an example of a style-conformant configuration of the TLS composed of two traffic lights (i.e., `TL`). It could represent the model of a crossroad with two traffic lights.

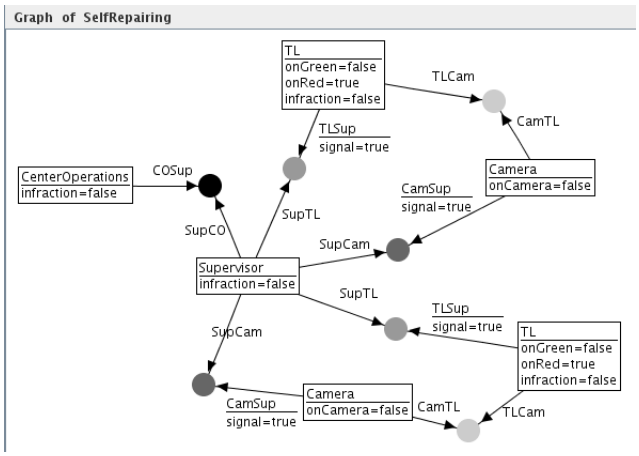

Figure 3. A configuration of TLS

B. TLS Productions

The set PR of productions of G_{TLS} models the following system configurations:

- *Switching*: the system switches between green/red light based on the traffic flow;
- *Light Check*: the system checks that each cross road has only one green light turned on;
- *Infraction management*: whether there is an infraction the system has to trigger the camera;
- *Loss of traffic light signal*: if there is a loss of a traffic light signal, the system tries to repair it;
- *Loss of Camera signal*: if there is a loss of `Camera` signal, the system tries to repair it.

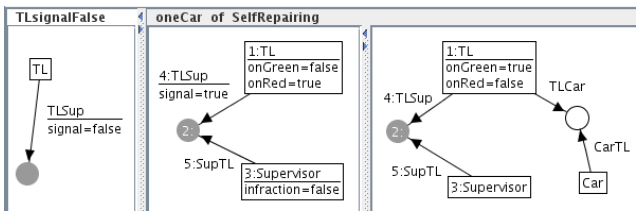

Figure 4. Rule *oneCar* of TLS

As shown in Sect. III the set of possible productions is composed of three subsets: Programmed (PR_{pgm}), Environment (PR_{env}) and Repairing (PR_{rpr}). In the following we describe only one production for each subset and remand the reader to see the complete case study at the following link[3]. The production presented in Fig. 4 shows the situation in which a `Car` component is added to the TLS. This means that the biggest amount of traffic is in that specific `TL`. Therefore, the addition of a `Car` means switching the `TL` attribute *onGreen* from false to true, and the attributes *onRed* from true to false. Attributes of the components in the left hand-side are set to values required for this situation. This rule cannot be applied when there is another `TL` such that $onGreen{==}true$. Additionally, the NAC *TLsignalFalse* controls the rule application by preventing that the rule is applied when at least one `TL` component does not send a signal. The left hand-side production presented in Fig. 5 models the situation in which the system changes its status from a desirable one into an undesirable one. Here, the system is in a status where the signal emitted by `TL` is correctly received and it goes to a status where the signal from `TL` is lost. Then, the component `TL` cannot communicate with the rest of the system. The right hand-side instead models the situation in which the system goes from an undesirable status in which the signal from `TL` is lost, to a desirable one, where the signal is re-established.

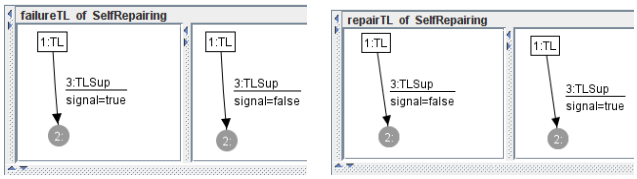

Figure 5. Rules *failureTL* and *repairTL* of TLS

VI. TLS Validation Using AGG

In this section we demonstrate how the analysis feature of AGG can be used for self-repairing system specifications in order to check it for correctness and completeness. In the following we start briefly describing how the TLS is implemented in AGG. Then we show how AGG is used to validate the system specification.

A. TLS Implementation in AGG

The TLS is represented by a *Typed Attributed Graph Grammar*. In the Type Graph (Style) shown in Fig. 2, each component of the system is represented by a hyperedge with a label. Each tentacle, represented by an edge, represents each single port of communication. Moreover, nodes and edges are attributed and multiplicity constraints are defined.

The productions of the TLS are *Typed Attributed Rules* in AGG. Most rules of the TLS are equipped with one or more NAC. Attributes, multiplicities, and NACs are very useful, in addition to control the application of rules, especially for expressing consistency conditions of the TLS. Finally, we defined some graph consistency constraints for the grammar. They specify global properties of the TLS which must be satisfied to guarantee the validity of the system.

[3]www.antoniobucchiarone.it/TLSCaseStudy/WICSA09.zip

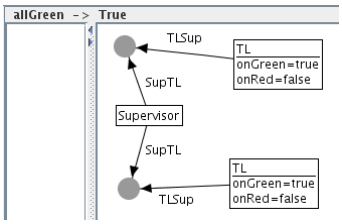

Figure 6. Graph consistency constraint *allGreen* of TLS

Figure 6 shows an atomic graph constraint, called $allGreen$, which is used in a Boolean constraint, called $oneGreen$. $oneGreen$, in the form of $\neg allGreen$, is defined in order to forbid any configuration that has two contemporary TLs such that $onGreen{=}true$ and $onRed{=}false$. The premise of this graph constraint (and of all other) is given by an empty graph which is always found in the start and each transformed graph. Once the conclusion is found, the Boolean constraint *oneGreen* fails and then it prevents a rule application which causes an inconsistent configuration. Table I summarizes the aspects of TLS formalization and its corresponding implementation in AGG.

Informal Aspect	Formalization	AGG Implementation
Comp and Conn	Hyperedge	Nodes and Edges
Port and Role	Hyperedge Tentacles	Edge with Node
SA Configuration	Hypergraph	Directed Graph
Comp/Conn Connections	Node	Node
Architectural Style	Hypergraph	Type Graph
Style-compliant SA configuration	Typed Hypergraph	Typed Attributed Graph
SA reconfiguration	Partial, injective morphism of T-typed hypergraphs	Partial, injective typed attributed rules
DSA	T-typed hypergraph grammar	Typed Attributed Graph Grammar

Table I
RELATIONS BETWEEN FORMALIZATION AND AGG IMPLEMENTATION

B. Repairing dynamism validation

The formalization of *completeness* and *correctness* of a repairing dynamism specification is explained in Sect. III.

The verification of *completeness* can be performed in AGG by means of *critical pair analysis*. More precisely, if for each pair of rules (p_1, p_2) with $p_1 \in PR_{pgm}$ and $p_2 \in PR_{rpr}$ no dependency can be found, then the normal (programmed) rules do not enable application of repairing rules. This means, if a repairing rule could be applied, then the configuration before was incorrect, so any programmed rules could not be applied, but only repairing. After repairing rule is applied the configuration will be correct again and a programmed rule is applicable. The dependency analysis results in Fig. 7 show that the dependency (*change - use attribute*) exists for each pair of rules (p_1, p_2) with $p_1 \in PR_{rpr}$ and $p_2 \in PR_{pgm}$. When there is no dependency of a repairing and programmed rules then we have to ask what will repair such a rule and have to check it. Conflicting rule pairs in Fig. 8 show that each repairing rule is in conflict with itself because a repairing rule prevents the applicability of itself.

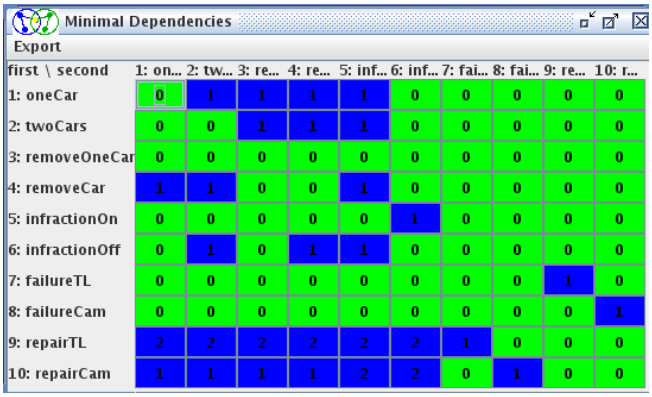

Figure 7. Dependency Matrix of TLS in AGG

The verification of *correctness* can be described in terms of *critical pair analysis* as following: for each pair of rules (p_1, p_2), with $p_1{\in}PR_{env}$ and $p_2{\in}PR_{rpr}$ there must exist a dependency (i.e., *change - use attribute*). This means that after application of an environment rule, which destroys a consistent configuration, an opposite repairing rule is applicable to restore consistency of this configuration. We can see these dependences in Fig. 7. Moreover, Fig. 8 shows that each environment rule is in conflict with itself since an environment rule prevents applicability of itself. It is important to note that, each pair (p_1, p_2) whit $p_1{\in}P_{env}$ and $p_2{\in}P_{pgm}$ are in conflict because of nature of *environment*. For example, once the rule *failureTL* is applied, the signal of TL component is lost and it cannot communicate with the rest of the system. The traffic will stop and wait for the rule *repairTL*, which repairs invalid configuration and enables application of normal (programmed) rules.

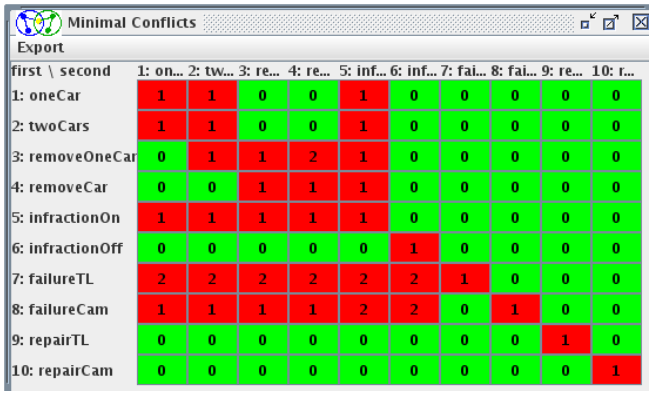

Figure 8. Conflict Matrix of TLS in AGG

The results of the verification of *critical pairs* performed in AGG are shown in Fig. 7 and in Fig. 8. In Fig. 7, the colored fields represent dependencies, while the colored

fields of the table in Fig. 8 represent the conflicts. The numbers in the cells mean the number of critical situations (overlapping graphs) of each rule pair.

As an example of the performed analysis, let us consider the critical rule pair *(failureTL, oneCar)*. Figure 9 shows a conflict situation (bold black colored *TLSup* edge with the attribute $signal{=}true$). The rule *failureTL* changes the attribute value from *true* to *false* and this impedes the application of the rule *oneCar* since it needs this attribute set to *true*.

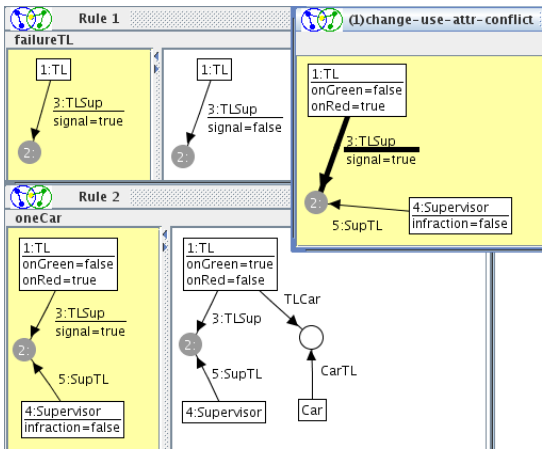

Figure 9. Conflict of rules *(failureTL, oneCar)*

Figure 10 shows a dependency situation of the rules *(repairTL, oneCar)*. Rule *repairTL* changes the attribute value from *false* to *true* and this restores the configuration in which the rule *oneCar* is applicable again.

Analyzing critical pairs which represent analogous situations we can see that all conflicts which are caused by the environment rules (*failureTL*, *failureCam*) will be resolved by repairing rules (*repairTL*, *repairCam*).

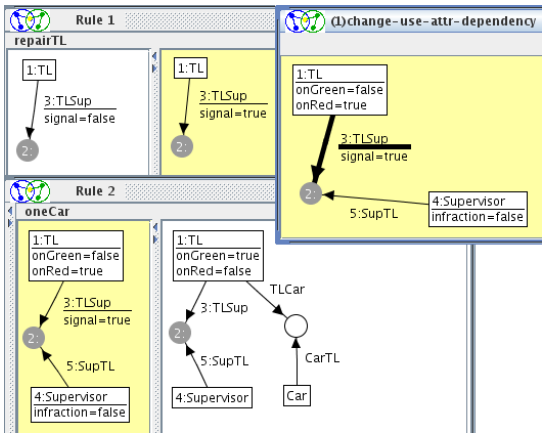

Figure 10. Dependency of rules *(repairTL, oneCar)*

C. Self dynamism verification

The formalization of completeness and correctness in context of self dynamism of DSA is shown in Sect. III-B. We recall that $\mathcal{S}(\mathcal{G}_\mathcal{S})$ and $\mathcal{E}_c(\mathcal{G}_\mathcal{S})$ are subsets of $\mathcal{R}(\mathcal{G}_\mathcal{S})$, where $\mathcal{S}(\mathcal{G}_\mathcal{S})$ are self reconfigurations by applying autonomous changes, $\mathcal{E}_c(\mathcal{G}_\mathcal{S})$ are reconfigurations by applying an external sequence of commands, and $\mathcal{R}(\mathcal{G}_\mathcal{S})$ is the set of all reachable configurations.

In our case study the TLS non-deterministically selects a rule to be applied. Therefore, the set $\mathcal{S}(\mathcal{G}_\mathcal{S})$ is the set of all configurations reachable by applying autonomous changes. Following, it can be associated to our TLS grammar which defines self dynamism with autonomous repairing mechanism. The set $\mathcal{E}_c(\mathcal{G}_\mathcal{S})$ is empty. The verification of *correctness* of the $\mathcal{S}(\mathcal{G}_\mathcal{S})$ using *critical pair analysis* can be reduced to check if for each pair (p_1, p_2) whit $p_1{\in}PR_{pgm}$ and $p_2{\in}PR_{rpr}$ no dependencies exist. Considering the critical pairs for TLS in Fig. 7, we do not find any dependency rule pair with the first rule from the set PR_{pgm} ($\{$*oneCar*,. . .,*infractionOff*$\}$) and the second from the set PR_{rpr} ($\{$*repairTL*, *repairCam*$\}$). In other words, each production of self reconfiguration does not produce configurations that need to be repaired.

The verification of *completeness* in the context of self dynamism is similar to the verification of *completeness* performed for repairing dynamism, as described in Sect. VI-B.

VII. Conclusion

In this paper we have presented a way to model self-repairing systems using Dynamic Software Architectures (DSAs) formalized as T-typed hypergraph grammars and implemented in a unique framework called AGG. AGG has been used to verify correctness and completeness of these models in an easy way. Future work concerns the extension of the modeling approach to model and analyze hierarchical DSAs that have as basic elements more complex and structured components. We plan to use hierarchical hypergraphs [26] in which each hyperedge can represent relations among components. Moreover we have in mind to consider also timing aspects in our formalization. In this paper we have not taken into account the behavior of components and connectors. As future work we will investigate mechanisms to support the adaptation of the system behavior. This research line will build on previous results: (i) [27] that presents an approach to efficiently (and still maintaining the required properties) manage the whole reconfiguration of the system when one or more components need to be updated; (ii) [28] that proposes a theoretical assume-guarantee framework that allows one to efficiently define under which conditions adaptation can be performed by still preserving the desired invariant.

ACKNOWLEDGMENTS

This work is partly supported by the Italian PRIN d-ASAP and from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement 215483 (S-Cube) projects.

REFERENCES

[1] B. H. C. Cheng, H. Giese, P. Inverardi, J. Magee, and R. de Lemos et al., "Software Engineering for Self-Adaptive Systems: A Research Road Map," in *Software Engineering for Self-Adaptive Systems*, ser. Dagstuhl Seminar Proceedings, no. 08031, 2008.

[2] R. Allen, R. Douence, and D. Garlan, "Specifying and analyzing dynamic software architectures," in *FASE'98*, 1998.

[3] J. Andersson, "Issues in dynamic software architectures," in *Int. Software Architecture Workshop*, 2000, pp. 111–114.

[4] L. Baresi, R. Heckel, S. Thone, and D. Varro, "Style-based refinement of dynamic software architectures," in *WICSA'04*. Washington, DC, USA: IEEE Computer Society, 2004.

[5] M. Hadj Kacem, M. Jmaiel, A. Hadj Kacem, and K. Drira, "Evaluation and comparison of adl based approaches for the description of dynamic of software architectures," in *ICEIS'05*. USA: INSTICC Press, 2005, pp. 189–195.

[6] D. Garlan and B. Schmerl, "Model-based adaptation for self-healing systems," in *WOSS '02*. ACM, 2002, pp. 27–32.

[7] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self-adaptation with reusable infrastructure," *Computer*, vol. 37, no. 10, pp. 46–54, 2004.

[8] J. S. Bradbury, J. R. Cordy, J. Dingel, and M. Wermelinger, "A survey of self-management in dynamic software architecture specifications," in *WOSS '04*. ACM, 2004, pp. 28–33.

[9] M. Wermelinger, "Towards a chemical model for software architecture reconfiguration," in *CDS '98*. IEEE Computer Society, 1998, p. 111.

[10] R. Bruni, A. Bucchiarone, S. Gnesi, and H. Melgratti, "Modelling dynamic software architectures using typed graph grammars," *ENTCS*, vol. 213, no. 1, pp. 39–53, 2008.

[11] R. Bruni, A. Bucchiarone, S. Gnesi, D. Hirsch, and A. Lluch Lafuente, "Graph-based design and analysis of dynamic software architectures." Berlin, Heidelberg: Springer-Verlag, 2008, pp. 37–56.

[12] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[13] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, *All About Maude - A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic*. LNCS, 2007.

[14] R. Morrison, D. Balasubramaniam, F. Oquendo, B. Warboys, and R. M. Greenwood, "An active architecture approach to dynamic systems co-evolution," in *ECSA*, 2007, pp. 2–10.

[15] R. Chatley, S. Eisenbach, J. Kramer, J. Magee, and S. Uchitel, "Predictable dynamic plugin systems," in *FASE*, 2004.

[16] D. Hirsch, J. Kramer, J. Magee, and S. Uchitel, "Modes for software architectures." in *EWSA'06*, 2006.

[17] M. Endler, "A language for implementing generic dynamic reconfigurations of distributed programs," in *BSCN'94*, 1994.

[18] P. Koopman, "Elements of the self-healing system problem space," in *WADS03*, 2003.

[19] S.-W. Cheng, D. Garlan, and B. Schmerl, "Architecture-based self-adaptation in the presence of multiple objectives," in *SEAMS'06*. New York, NY, USA: ACM, 2006, pp. 2–8.

[20] M. Autili, P. Inverardi, and P. Pelliccione, "Graphical Scenarios for Specifying Temporal Properties: an Automated Approach," *Automated Software Engineering (ASE)*, vol. 14, no. 3, pp. 293–340, September 2007.

[21] A. Bucchiarone, "Dynamic software architectures for global computing systems," Ph.D. dissertation, IMT Institute for Advanced Studies, Lucca, Italy, 2008.

[22] A. Habel, R. Heckel, and G. Taentzer, "Graph grammars with negative application conditions," *Fundamenta Informaticae*, vol. 26, pp. 287–313, 1996.

[23] R. Heckel and A. Wagner, "Ensuring consistency of conditional graph rewriting - a constructive approach," *Electr. Notes Theor. Comput. Sci.*, vol. 2, 1995.

[24] J. H. Hausmann, R. Heckel, and G. Taentzer, "Detection of conflicting functional requirements in a use case-driven approach," in *ICSE 2002*. ACM Press, 2002, pp. 105–115.

[25] L. Lambers, H. Ehrig, and F. Orejas, "Efficient conflict detection in graph transformation systems by essential critical pairs," *Electron. Notes Theor. Comput. Sci.*, vol. 211, pp. 17–26, 2008.

[26] B. H. F. Drewes and D. Plump, "Hierarchical graph transformation," *J. Comput. Syst. Sci.*, vol. 64, pp. 249–283, 2002.

[27] P. Pelliccione, M. Tivoli, A. Bucchiarone, and A. Polini, "An architectural approach to the correct and automatic assembly of evolving component-based systems," *J. Syst. Softw.*, vol. 81, no. 12, pp. 2237–2251, 2008.

[28] P. Inverardi, P. Pelliccione, and M. Tivoli, "Towards an assume-guarantee theory for adaptable systems," in *SEAMS2009*, 2009.