Modelling Dynamic Software Architectures using Typed Graph Grammars

Antonio Bucchiarone and Hernán Melgratti

IMT Lucca, Italy
{*a.bucchiarone,h.melgratti*}*@imtlucca.it*

and

Stefania Gnesi

ISTI-CNR of Pisa, Italy
stefania.gnesi@isti.cnr.it

and

Roberto Bruni

Università di Pisa, Italy
bruni@di.unipi.it

Abstract

Several recent research efforts have focused on the dynamic aspects of software architectures providing suitable models and techniques for handling the run-time modification of the structure of a system. A large number of heterogeneous proposals for addressing dynamic architectures at many different levels of abstraction have been provided, such as programmable, ad-hoc, self-healing and self-repairing among others. It is then important to have a clear picture of the relations among these proposals by formulating them into a uniform framework. Our work is a contribution in this line. In particular, we map several notions of dynamicity into the same formal framework (i.e. typed graph grammar) in order to distill the similarities and differences among them. As a result we provide a characterization of different styles of architectural dynamisms in term of graph grammars with the purpose to understand the kinds of properties that can be naturally associated to such different specification styles and we describe them over a simple case study.

Key words: Dynamic Software Architectures, Typed Graph Grammars and Modelling.

During the last decades, computer systems have changed from isolated static devices to highly interconnected machines that execute their tasks in a cooperative and coordinated manner. These modern, complex distributed systems are known as *global computing systems* (GCS) or *network-aware computers*, and have to deal with frequent changes of the network environment. In a GCS, components are autonomous, dynamic and often ad-hoc, the network's coverage is variable, and there is not a centralized authority. Software architectural models, which are intended to describe the structure of a system in terms of computational components, their interactions, and its composition patterns [22], allow to reason about systems at a more abstract level, disregarding implementation details. Since GCS may change at design time, pre-execution time, or run-time [19], software architecture models for GCS should be able to describe the changes of the system structure and to enact the modifications during the system execution [17]. Such models are generally referred to as *Dynamic Software Architectures (*DSA*s)*, to emphasize that the system architecture evolves during runtime. Nevertheless, a variety of definitions of dynamicity for software architecture have been proposed in the literature. Below we list some of the most prominent definitions to show the variability of connotations that the word *dynamic* acquires.

- **Ad-hoc dynamism** [7]. Modifications 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 dynamism** [2]. It is a kind of ad-hoc mechanism but there is a modification language for describing architectural changes.
- **Programmed Dynamism** [7]. Changes are triggered by the system and changes are defined prior to run-time.
- **Self-repairing** [21]. Changes are initiated and assessed internally, i.e., the run-time behavior of the system is monitored to determine whether a change is needed. In such case, a reconfiguration is automatically performed.
- **Self-adaptive** [20]. Systems can adapt to their environments by enacting run-time changes.

The different proposals for DSA are bound to particular languages and models. In this paper we are aimed at understanding the main notions relying behind such proposals by abstracting away from particular languages and notations. We want to give a uniform formal presentation that is abstract enough to cover most of those features. In this sense, our work is in the line of other previous research efforts [23,6]. In particular we select graph grammars as a formal framework for mapping the different notions of dynamicity because (i) they provide both a formal basis and a graphical representation that is in line with the usual way architectures are represented, (ii) they allows for a natural way of describing styles and configurations, (iii) they have been largely used for specifying architectures. Nevertheless, this work is aimed neither at providing a particular specification/programming language nor at promoting graph grammars as a modelling language for DSA. We just use this framework to compare different mechanisms and to understand the kinds of properties that can be naturally associated to such specifications. Hence, we argue that the characterisation of dynamicity we present is to some extend orthogonal to the particular kind of graph grammars we use, and therefore, extensible to different variants of graph rewriting systems.

Related Work. Several previous works have proposed alternative ways for describing software architecture (SA) by using graph grammar. Our representation of SA as graph grammars has been borrowed from the Le Métayer approach [18]. Actually the notion of programmable DSA corresponds to that proposal. A different way of representing SA with graphs can be found in [11], where hyperedges are components and nodes are ports of communication, and the reconfiguration is given as context-free productions together with a constraint solving mechanism. Also Baresi et al. in [3,4] use graph transformation systems to model programmed architectural styles at different levels of abstraction, and represent reconfiguration scenarios as graph transformation sequences.

As far as the different flavours of dynamicity are concerned, the work of Wermelinger in [23] explores the ability of the Chemical Abstract Machine (CHAM) [5] to express the dynamics of the architecture. His formalization tackles self-organized, ad-hoc, and programmed reconfiguration.

Ad-hoc reconfiguration has been studied in [7] as a programming language that allows for the runtime modification of SA. Similarly, proposal for constructible languages can be found in [19]. Self-repairing mechanisms have been proposed in [1,8,9,21]. These approaches are interested in providing executable frameworks for supporting DSA. The main difference of our work w.r.t. the previous proposals is that the latter are aimed at providing real specification/ programming/ implementation languages while we are aimed at giving an abstract characterization of such kind of mechanisms. The closest work to our proposal is [23], which proposes particular CHAM (and commands) for introducing changes. Differently, we are interested in understanding how each particular style of dynamism is reflected into a graph grammar. So, our main contribution is the identification of some features of graph rewriting systems that reveals a particular style of dynamism.

1 Formalization of Dynamicity

In this section we describe the formal framework we will use in the rest of the paper, and the way in which software architectures will be represented by using hypergraphs. In our formalization components and connectors are represented as hyperedges while ports are the nodes of the hypergraph. Figure 1 depicts an hypergraph containing two nodes port_1 and port_2, the hyperarc $\mathsf{component}$ standing for a component that exposes two different ports, and the hyperarc $\mathsf{connector}$ representing a connector that has two tentacles to the port port_1 and one to the port port_2. Note that we graphically depict components as square boxes and connectors by rounded boxes. Moreover, we show the ordering of the attach points of a hyperarc by labeling the corresponding arrows with natural numbers (we will also label arrows with arbitrary names instead of numbers when this facilitates the reading).

Definition 1.1 [Hypergraph]A *(hyper)graph* is a triple $H = (N_H, E_H, \phi_H)$, where

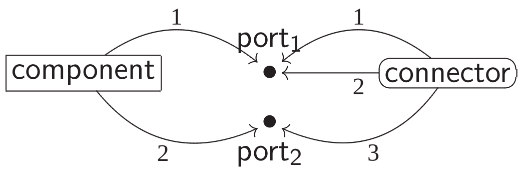

Figure 1. A hypergraph describing a style.

N_H is the set of nodes, E_H is the set of (hyper)edges, and $\phi_H : E_H \to N_H^+$ describes the connections of the graph, where N_H^+ stands for the set of non-empty strings of elements of N_H. We call $|\phi_H(e)|$ the *rank* of e and assume that $|\phi_H(e)| > 0$ for any $e \in E_H$.

The connection function ϕ_H associates each hyperedge e to an ordered, non empty sequence of nodes, i.e., the nodes to which e is attached to. Given a hypergraph T describing an architectural style, i.e., a hypergraph that describes only the types of ports, connectors, components and the allowed connections, a configuration compliant to such style will be described by the notion of T-typed hypergraphs introduced below:

Definition 1.2 [Typed Hypergraph] Given a hypergraph T (called the *style*), a T-*typed hypergraph* or *configuration* is a pair $\langle |G|, \tau_G \rangle$, where $|G|$ is the *underlying* graph and $\tau_G : |G| \to T$ is a total hypergraph morphism.

We recall that a total hypergraph morphism $f : G \to G'$ is a couple $f = \langle f_N : N \to N', f_E : E \to E' \rangle$ such that: $f_N(\phi_G(e)) = \phi_{G'}(f_E(e))$ (We apply f_N over strings by meaning the homomorphic extension of f_N over strings).

The graph $|G|$ defines the configuration of the system, while τ_G defines the (static) *typing* of the resources. Consider the graph T introduced in Figure 1 as a style stating that there is one unique type component of components exposing two ports of different types, and one connector attached to two ports of type port$_1$ and one port of type port$_2$. Then, a possible T-typed hypergraph (or a configuration of the style T) is shown in Figure 2. The typing morphism is implicitly defined by the name of the elements in the configuration, which consist of the type name plus a subindex identifying the particular instance. For instance, the port port$_{1A}$ has the type port$_1$. The configuration in Figure 2 has two different components of type component with their corresponding ports, and one connector of type connector. We remark that the typing morphism requires components to have exactly one port of type port$_1$ and one of type port$_2$. Similarly, the only connections valid for a connector are those that attach two arcs to ports of type port$_1$ and one to a port of type port$_2$. All such constraints are enforced by the existence of a typing morphism.

Finally, the reconfiguration of a software architecture is described by a set of rewriting productions. Roughly, a production p is a partial, injective morphism of T-typed graphs, i.e., it has the following shape: $p : L \rightarrowtail R$, where L and R are T-typed hypergraphs that are called the *left-hand* and the *right-hand side* of the production, respectively. Given a T-typed graph G and a production p, a rewriting of G using p can be informally described as follow:

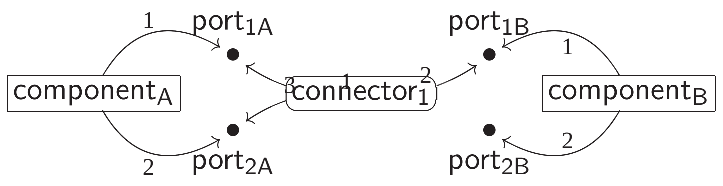

Figure 2. A hypergraph describing a configuration for the style in Figure 1

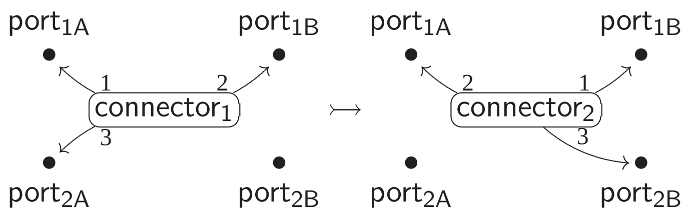

Figure 3. A rewriting production.

- Find a (type preserving) match of the left-hand-side L in G, i.e., identify a sub-graph of G that corresponds with L,
- Remove from the graph G all the items corresponding to the left-hand side that are not in the right-hand-side,
- Add all the items of the right-hand side that are not in the left-hand-side

The elements that are both in L and R are preserved by the rewriting step.

An example of a production is shown in Figure 3 (the morphism among the left and right-hand side of the rule is represented by using the same names for mapped elements, i.e., it is the partial inclusion). The production allows to remove an existing connector $\mathsf{connector}_1$ and to add a new connector $\mathsf{connector}_2$ that is attached to the original ports in a specular way with respect to the original connector.

An architecture will be described by a T-typed graph grammar.

Definition 1.3 [(T-typed) graph grammar] A *(T-typed) graph grammar* $\mathcal{G}$ is a tuple $\langle T, G_{in}, P\rangle$, where G_{in} is the *initial (T-typed) graph* and P is a set of *productions*.

Notation. Let $\mathcal{G} = \langle T, G_{in}, P\rangle$ be a (T-typed) graph grammar, and G and H (T-typed) hypergraphs. We write $G \Rightarrow_p H$ to denote that G is rewritten in one step to H by using the production p. We abbreviate the reduction sequence $G_0 \Rightarrow_{p_0} G_1 \Rightarrow_{p_1} G_2 \ldots \Rightarrow_{p_n} G_{n+1}$ with $G_0 \Rightarrow_{p_0;p_1,\ldots;p_n} G_{n+1}$. We write $G \Rightarrow^* G'$ to denote that there exists a possible empty sequence $s \in P^*$ of derivation steps such that $G \Rightarrow_s G'$.

For convenience when describing the examples, we will also consider productions with negative application conditions [10], i.e., productions that are equipped with a constraint about the context in which they can be applied. For instance, such conditions can state that the production is applicable only when certain nodes, edges, or subgraphs are not present in the graph. Such conditions are graphically shown in the left-hand-side of a production by grouping forbidden elements into a dotted lined area. Figure 4 shows a production with negative conditions stating that the new connector $\mathsf{connector}_2$ can be added to the configuration if and only if no

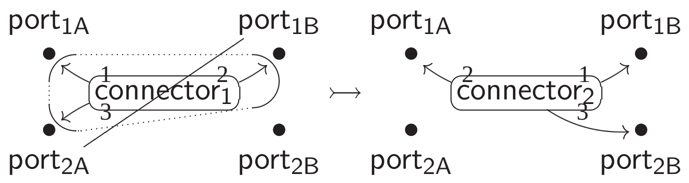

Figure 4. A rewriting production with negative application condition.

other connector of type connector is already attached in a specular way to port_{1A} and port_{1B}.

Remark. We refer the interested reader to the formal presentation of SPO graph grammars to [15] and to [10] for grammars with negative application conditions.

2 Characterisation of Dynamism

This section characterizes different forms of dynamism in software architecture in terms of graph grammars. In particular we consider the following four forms of dynamicity: Ad-hoc, Constructible, Repairing and Programmed [2,7,9,19,20,21].

Given a grammar $\mathcal{G} = \langle T, G_{in}, P \rangle$, we will use the following notions:

- The set $\mathcal{R}(\mathcal{G})$ of reachable configurations, i.e., all configurations to which the initial configuration G_{in} can evolve. Formally, $\mathcal{R}(\mathcal{G}) = \{G | G_{in} \Rightarrow^* G\}$.
- The set $\mathcal{D}_{\mathsf{P}}(\mathcal{G})$ of desirable configurations, i.e., the set of all T-typed configurations that satisfies a desired property P. In other words, the set of acceptable configurations of an architecture are (explicit or implicitly) defined as the graphs that have type T and satisfies a property P. Formally,
$$\mathcal{D}_{\mathsf{P}}(\mathcal{G}) = \{G \mid G \textit{ is a } T\textit{-typed graph} \wedge \mathsf{P} \textit{ holds in } G\}$$

2.1 Programmed dynamism

Programmed dynamism assumes that all architectural changes are identified at design time and triggered by the program itself [7]. Many proposals in the literature [18,12,4] that use graph grammars for specifying DSA present this kind of dynamism. A programmed DSA $\mathcal{A}$ is associated with a grammar $\mathcal{G}_{\mathcal{A}} = \langle T, G_{in}, P \rangle$, where T stands for the style of the architecture, G_{in} is the initial configuration, and the set of productions P gives the evolution of the architecture.

The grammar fixes the types of all elements in the architecture, and their possible connections. Moreover, the productions state the possible ways in which a configuration may change. Hence, programmed dynamism provides an implicit definition of desirable configurations. That is, assuming the specification to be complete and correct, the sets of desirable and reachable configurations should coincide, i.e., $\mathcal{D}_p(\mathcal{G}) = \mathcal{R}(\mathcal{G})$.

2.1.1 Programmed dynamism - Verification aspects

Programmed dynamism enables for the formulation of several verification questions. Consider the set of desirable configurations $\mathcal{D}_{\mathsf{P}}(\mathcal{G})$, then it should be possi-

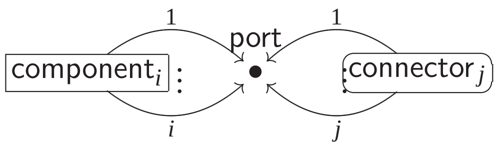

Figure 5. A general graph of types of a software architecture

ble (at least) to know whether:

- the specification is correct, in the sense that any reachable configuration is desirable. This reduces to prove that $\mathcal{R}(\mathcal{G}) \subseteq \mathcal{D}_{\mathsf{P}}(\mathcal{G})$, or equivalently that $\forall G \in \mathcal{R}(\mathcal{G}) : \mathsf{P}$ *holds in G*.
- the specification is complete, in the sense that any desirable configuration can be reached. This corresponds to prove $\mathcal{D}_{\mathsf{P}}(\mathcal{G}) \subseteq \mathcal{R}(\mathcal{G})$, or equivalently that *if* P *holds in G then* $G \in \mathcal{R}(\mathcal{G})$.

2.2 Ad-hoc dynamism

Roughly ad-hoc dynamism allows the architecture to evolve freely by adding and removing components and connectors without any restriction. We describe an ad-hoc DSA $\mathcal{A}$ with a typed grammar $\mathcal{G}_{\mathcal{A}} = \langle T^{\mathrm{ah}}, G_{in}, P^{\mathrm{ah}} \rangle$, where

- The graph of types T^{ah} is the graph in Figure 5 that contains an infinite number of hyperarcs $\mathsf{component}_i$ and $\mathsf{connector}_j$, one for every natural $i, j \in \mathbb{N}$. Any hyperarc $\mathsf{component}_i$ ($\mathsf{connector}_j$) stands for the type of all connectors that expose exactly i ports (respectively, j roles). Moreover, the function $\phi_{T^{\mathrm{ah}}}$ associates every attachment of an hyperarc (either $\mathsf{component}_i$ or $\mathsf{connector}_j$) to the unique node port. For simplicity, we define all ports as having the same type. Nevertheless, the type graph can be extended, by adding an infinite number of nodes, to represent every possible type of port.
- The initial graph G_{in} is any T^{ah}-typed hypergraph that describes a *software configuration*, i.e., no isolated nodes (or ports) and ports having attached at most one hyperarc corresponding to a component. This is the only one restriction imposed on configurations by ad-hoc dynamism.
- The set of production P^{ah} is infinite. Roughly, they can be the combination of the following four actions: (i) the creation of a new component, i.e., the addition of a new hyperarc of type $\mathsf{component}_i$ and the generation of its i associated nodes of type port; (ii) the removal of a component and all its associated ports; (iii) the addition of a connector, i.e., the creation of a new hyperarc of type $\mathsf{connector}_j$ attached to j existing nodes of type port; and (iv) the removal of a connector.

Since an ad-hoc DSA allows for any kind of changes in the configuration, the set $\mathcal{R}(G_{\mathcal{A}})$ of reachable configurations is the set of all possible (T^{ah}-typed) hypergraphs describing a software architecture. The only guaranties given by ad-hoc dynamicity is that reached graphs are software configurations. However, it does not provide any restriction to the set of reachable configurations. Consequently, an ad-hoc architecture provides no information about desirable or admissible configurations, and hence, there is not a notion of style that should be enforced through the reconfiguration. Moreover, a specific ad-hoc DSA is characterized only by its initial configuration G_{in}, because the type graph and the productions are the same for all ad-hoc DSAs.

2.2.1 Ad hoc dynamism - Verification aspects

The kind of verification questions formulated for programmable DSAs are meaningless for the case of ad-hoc dynamism. Although this feature make ad-hoc dynamicity useless for specifying architectures, we remark that they were proposed for handling the run-time reconfiguration of systems in which changes are controlled by the environment (basically a programmer). We postpone the discussion of autonomous and externally controlled changes until Section 4.

2.3 Constructible dynamism

Constructible DSAs are similar to ad-hoc DSAs but here rewriting productions are not the free combination of basic primitives: they are full-fledged programs written in some specific language. The main difference w.r.t. ad-hoc DSA is that a constructible dynamic architecture is not only characterised by its configuration but also by the specific programming language allowed for defining the reconfiguration programs that can manage the evolution. Let $\mathcal{L}_c$ be the language for writing reconfiguration programs, and $P^{\mathcal{L}_c}$ the set of all valid reconfiguration programs (or graph productions) that can be written in $\mathcal{L}_c$, a constructible DSA $\mathcal{A}$ is described by $\mathcal{G}_{\mathcal{A}} = \langle T^{\mathrm{ah}}, G_{in}, P^{\mathcal{L}_c}\rangle$, where the type graph T^{ah} and the initial configuration G_{in} are as for ad-hoc dynamism.

Fixed a language $\mathcal{L}_c$, the relation among ad-hoc and constructible dynamism can be stated by the following relation among reachable configurations:

$$\mathcal{R}(\langle T^{\mathrm{ah}}, G_{in}, P^{\mathcal{L}_c}\rangle) \subseteq \mathcal{R}(\langle T^{\mathrm{ah}}, G_{in}, P^{\mathrm{ah}}\rangle)$$

Although the graphs of types in both ad-hoc and constructible DSAs give no information about the architectural style of reachable configurations and, in principle, a constructible DSA may be specified by allowing any possible configuration to be reached, it could be also the case that the reconfiguration language $\mathcal{L}_c$ forbids some transformations (e.g., not valid programs). In this case, a constructible DSA may provide an implicit, weak notion of desirable configurations.

2.3.1 Constructible dynamism - Verification aspects

We remark that constructible dynamism is intended to handle run-time reconfiguration, and hence the verification of correctness and completeness properties shares similarities with ad hoc dynamism. Generally speaking, constructible dynamism provides a very weak notion of desirable configurations, and hence the verification of such properties when assuming autonomous reconfiguration is almost meaningless. However, the situation is different when considering reconfigurations controlled externally (see discussion in Section 4).

2.4 *Repairing (or healing) dynamism*

Self repairing systems are 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 the configuration [8].

We can think about a repairing architecture as an ordinary graph grammar $\mathcal{G}_{\mathcal{A}} = \langle T, G_{in}, P\rangle$ in which the set of productions is divided into three different sets, i.e., $P = P_{pgm} \cup P_{env} \cup P_{rpr}$. Rules in P_{pgm} describe the normal, ideal behaviour of the architecture, i.e., $\mathcal{G}'_{\mathcal{A}} = \langle T, G_{in}, P_{pgm}\rangle$ is a programmed DSA. Rules in P_{env} model the *environment* or, in other words, the ways in which the behaviour of the architecture may deviate from the expected one. Rules in P_{env} may state that the communication among components may be lost or that a non authorised connector become attached to a particular component. Rules P_{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 P_{rpr} identifies a composition pattern in the system that is undesirable. In this way a repairing architecture implicitly defines the desirable configurations of the system:

$$\begin{aligned} G \in \mathcal{D}_{\mathsf{P}}(\mathcal{G}_{\mathcal{A}}) \quad \textit{iff} \quad & G \in \mathcal{R}(\mathcal{G}_{\mathcal{A}}) \wedge \\ & \neg(\exists q \in P_{rpr}, \exists G' \in \mathcal{R}(\mathcal{G}_{\mathcal{A}}) : G \Rightarrow_q G') \end{aligned}$$

In other words, G is a desirable configuration if and only G is a reachable configuration that does not exhibit an undesirable composition pattern (i.e.,a left-hand-side match for a repairing rule).

2.4.1 *Repairing dynamism - Verification aspects*

As for the case of programmable dynamism, repairing dynamism allows for the formulation the following two questions:

- the specification is complete. This reduces to prove that $G \in \mathcal{D}_{\mathsf{P}}(\mathcal{G}_{\mathcal{A}})$ implies $G \in \mathcal{R}(\mathcal{G}_{\mathcal{A}}) \wedge \neg(\exists q \in P_{rpr}, \exists G' \in \mathcal{R}(\mathcal{G}_{\mathcal{A}}) : G \Rightarrow_q G')$.
- the specification is correct. This corresponds to prove $G \in \mathcal{R}(\mathcal{G}_{\mathcal{A}}) \wedge \neg(\exists q \in P_{rpr}, \exists G' \in \mathcal{R}(\mathcal{G}_{\mathcal{A}}) : G \Rightarrow_q G')$ implies $G \in \mathcal{D}_{\mathsf{P}}(\mathcal{G}_{\mathcal{A}})$.

In addition, this kind of dynamism naturally poses the question of whether reparing rules are adequate, i.e., whether the set of reparing rules assures that for any configuration that is reachable but not desirable there exists a sequence of repairing rules that moves the configuration to a desirable one. Formally,

- If $G \in \mathcal{R}(\mathcal{G}_{\mathcal{A}}) \wedge (\exists q \in P_{rpr}, \exists G' \in \mathcal{R}(\mathcal{G}_{\mathcal{A}}) : G \Rightarrow_q G')$ then $G \Rightarrow_{q_0} G_1 \Rightarrow_{q_1} \ldots \Rightarrow_{q_n} G_n$ with $G_n \in \mathcal{D}_{\mathsf{P}}(\mathcal{G}_{\mathcal{A}})$ and $\{q_0, \ldots, q_n\} \in P_{rpr}$.

3 Automotive Software System

In order to illustrate the different notions of dynamism presented before, we introduce the following scenario borrowed from the European Project SENSORIA concerning to the automotive case study.

3.1 Automotive Case Study: Overview

Much of the cost of research and development in vehicle production are associated to automotive software. This fact has increased the importance of software engineering concerns into the automotive domain. Today vehicles are equipped with a multitude of sensors and actuators that provide different services, like ABS and vehicle stabilization systems, that assist people to drive safer. Thanks to current mobile technology, vehicles have now the possibility to connect to the telephone and internet infrastructures. This has given birth to a variety of new services into the automotive domain. Communication in AS systems may involve *communication* that takes place inside a vehicle (*intra-vehicle)*, connection to vehicles in the vicinity (*inter-vehicle)*, or interaction with the environment, for example through an Internet gateway (*vehicle-environment)*. In the following scenario we will consider only the last two kinds of communications.

3.2 Car Assistance

Consider a vehicle subscribed to an assistance service. Due to a collision, the airbag of the car is inflated, which causes the automatically generation of a message destined to the *accident assistant server*. Once generated, the message can be transmitted through near vehicles until reaching the server (preferred method) or directly to the server. The message will be eventually delivered to the assistance server, which will coordinate the assistance. Consequently, we will model the architecture of the AS system by using two different entities:

- **Vehicle (V)**: a component responsible for transmitting messages destined to the assistant server. A vehicle component has three associated ports: p_{in} for receiving a message from another vehicle, p_{out} for sending messages to the next vehicle, and p_s for communicating directly with the server.
- **Accident Assistant Server (S)**: a component that handles help requests. Its unique port p_{io} is used for sending/receiving information to/from vehicles.

Components are connected by using the following connector types:

- **Vehicle to Vehicle communication (V/V)**: a connector used for mediating the communication between two vehicles.
- **Vehicle to Accident Assistant Server communication (V/S)**: a connector used for supporting the interaction between a vehicle and a server.

Figure 6 shows the architectural style of the AS system, while Figure 7 depicts an instance consisting of two vehicles (V1 and V2) and one server (S).

3.3 Programmed Dynamism

We will use a programmable architecture for specifying the way in which the AS system keeps the communication structure among the components. We define the corresponding graph grammar $\mathcal{G}_{\mathcal{A}} = \langle T, G_{in}, P\rangle$, where the architectural style T is

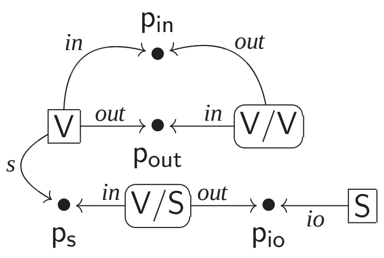

Figure 6. Architectural Style of the AS System

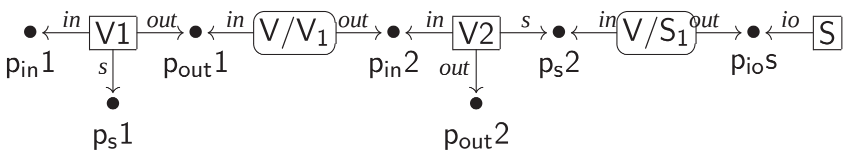

Figure 7. An instance of the AS style.

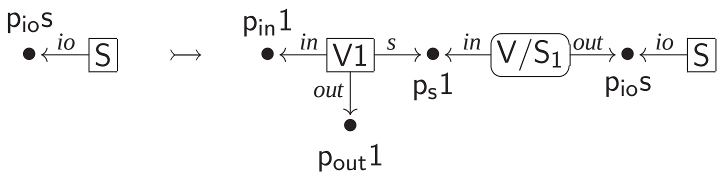

Figure 8. New Vehicle connected to the Server.

as depicted in Figure 6, and the initial configuration G_{in} consists in the T-typed graph containing a unique component of type server S and its associated port (i.e., a node of type $\mathsf{port}_{\mathsf{io}}$). The set P contains the productions that model the arrival/departure of a vehicle into/from the area covered by a server, a car that is getting close/far to/from another car, and a car that takes over another one. For space limitations, we describe just three productions. Figure 8 stands for the case of vehicle entering into the area covered by a server. This is modeled by creating a new component of type V and its associated ports, and a new connector between the car and the server. Figure 9 describes the case of a car connected to the server (V3) that gets closer to another vehicle (V2) that is at the end of a queue (note V2 has not connectors attached to its input port). In this case the connection of V3 to the server is removed and a new connector between V3 and V2 is created. Finally, Figure 10 depicts a production that handles the case in which a vehicle takes over another one. (Note the rule assumes the vehicles V1 and V2 to be in the middle of a queue, since they are connected to other vehicles. The complete specification should consider three additional rules to handle the cases in which the take over involves cars at the ends of the queue.).

The set of desirable configurations for the AS system consists of all the configurations in which each vehicle has a unique, acyclic communication path with the unique server, and each vehicle port has attached at most one connector.

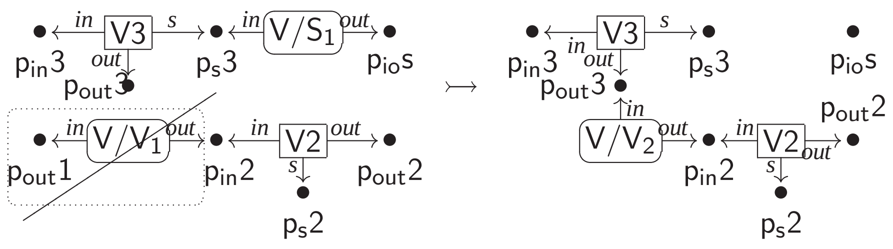

Figure 9. Vehicles approximation.

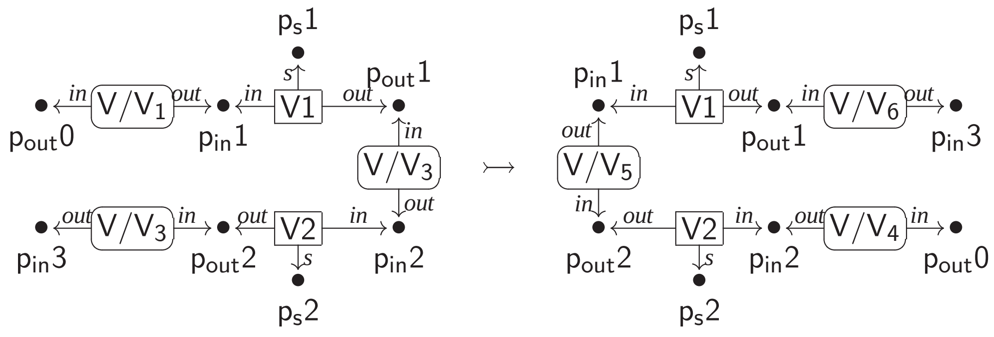

Figure 10. One vehicle takes over another one.

3.4 Ad-hoc dynamism

Ad-hoc changes are intended to handle run-time reconfigurations on-demand. Let us assume that the programmable architecture introduced previously has been deployed and now it is running. Consider that we desire to cover a larger area, and hence more vehicles. In this case we may need to reconfigure the architecture by simply adding new servers, or we can decide to move some queue to the newly created servers, or we may want to replace the old server by several copies of a new one, and so on. As discussed in Section 2, ad-hoc reconfiguration imposes no constraints on the kind of changes that can be made to the architecture. Hence, in principle, any kind of reconfiguration could be created (even elements with new types). This notion corresponds to the grammar introduced in Section 2, which is the model for any ad-hoc DSA (as already mentioned).

3.5 Constructible dynamism

A constructible DSA is, to some extent, a restriction of the ad-hoc mechanism, in which transformations should be written by using a particular language. For example, consider the language $\mathcal{L}_c$ that fixes a particular set of types: component types are *vehicle* (V) and *server* (S) whose ports are as defined previously, connector types are V/V and V/S, whose ports are as in the previous cases. Programs in $\mathcal{L}_c$ are built by using the following commands: *newC* to create a new component with its associated ports, *newK* to create a new connector attached to existing ports, *delK* to remove a connector, and *delCIF*, which is used to remove a component but only when it has no attached connectors. Note that in this case, we cannot write reconfiguration programs (or productions) that deletes a component that has an attached

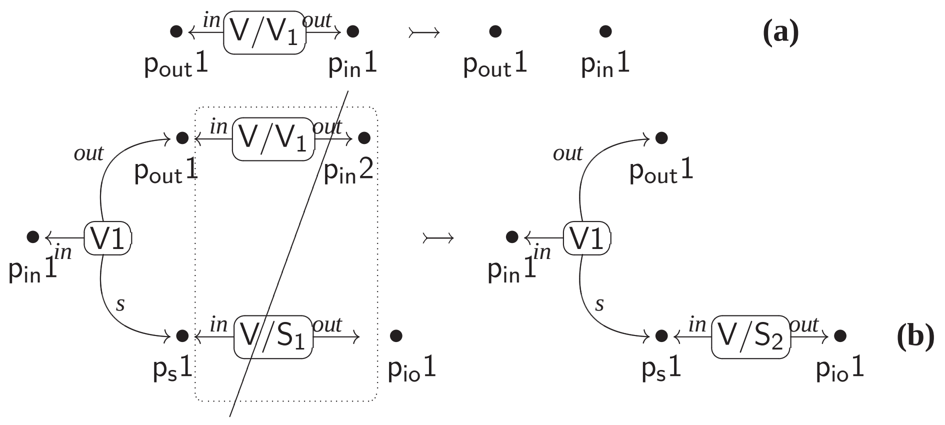

Figure 11. (a)Lost of connectivity. (b) Repairing.

connector. Note, that although the set $P^{\mathcal{L}_c}$ of programs that can be written in $\mathcal{L}_c$ is infinite, it is strictly included into the set of productions available in the ad-hoc architecture, i.e., $P^{\mathcal{L}_c} \subset P^{ah}$.

3.6 Repairing dynamism

The following example shows the use of a repairing architecture for modelling the fact that the communication between vehicles is not reliable and can be lost, but in such cases the architecture should repair itself in order to provide unconnected components with a link to a server. We defined a graph grammar $\mathcal{G}_{\mathcal{A}} = \langle T, G_{in}, P\rangle$ in which the set of productions is divided into three different sets, i.e., $P = P_{pgm} \cup P_{env} \cup P_{rpr}$. In our scenario P_{pgm} contains the same productions as defined in Programmed Dynamism (see 3.3), P_{env} contains the unique production shown in Figure 11.*a)*, which models the loss of connectivity between vehicles (i.e., the removal of a connector V/V_1), and repairing production is in Figure 11.*b)*. Such rule states that whenever a vehicle without outcoming connections is found (note the left-hand-side of the rule requires the absence of connections on ports $\mathsf{p_{out}1}$ and $\mathsf{p_s1}$), then the vehicle should be connected directly to a server.

As for the case of programmable DSAs, in desirable configurations all vehicles have a connection to the server. Nevertheless, the repairing grammar takes into account the cases in which the configuration may not satisfy this condition (some vehicles may not be connected to a server). However, these cases match with the left-hand-side of the repairing rule, and hence, they can be repaired by the system.

4 Constrained and Self dynamism

Other aspects that are, to some extent, orthogonal to the approaches characterised in Section 2 are: (i) whether the application of a transformation rule can take place at any moment or not, and (ii) whether changes are fired internally by the system or activated externally. The first aspect is usually refered to as *constrained vs unconstrained dynamism*, while the second is associate to the qualification *self* vs *external* reconfiguration.

4.1 Unconstrained vs Constrained dynamism

Basically, constrained dynamism refers to the fact that a change may occur only after pre-defined constrains are satisfied. Such constraints may be (i) the configuration topology, e.g., when components are not connected in a specific, or (ii) the state of a component, e.g., when a component enters into the quiescent state. Topological constraints are naturally modelled by both positive and negative application conditions of graph productions. Hence, topological constrained dynamism may be characterised by a graph grammar whose productions have contexts (either positive or negative).

Differently, constraints related to particular states of components have not an immediate counterpart in our proposal (since our framework does not describe component states). Nevertheless, they can be encoded by thinking about different states of components as different types of hyperarcs. In this way, the change of a component state s into s' is represented as the rewriting step that removes the hyperarc denoting the component in state s and adds a new hyperarc of type s' with attachments analogous to those of the removed arc. In this case, the fact that the grammar describes a dynamism constrained on the state of some components is hidden by the encoding. An alternative approach could be the use of attributed graph grammars [16] for associating any component with an attribute describing its state.

Unconstrained dynamism refers to the fact that transformations can be applied at any moment. The graph grammar counterpart is the fact that productions have no associated constraints or application conditions, being, in some sense, context free, because they either produce or consume arcs but they do not read them.

4.2 Self dynamism

Usually, some kind of dynamisms (particularly programmed, and repairing) are also qualified as "self", meaning that the changes are initiated by the system itself and not by an external agent. We map the notion of self and external dynamism to particular features of the rewriting system. As a starting point we discuss the different alternatives for chosing a particular reconfiguration in a DSA, as proposed in [6].

- *Explicit*: The reconfiguration rule is selected by an external source and not by the system itself. This option resembles the external choice of process calculi, in which the branch of computation to be selected is indicated by the context of process. In this sense, we can interpret a reduction of the form $G \Rightarrow_p G'$ as the fact that the environment selects the application of the production p, and hence the reduction can take place only when the environment choses to apply such rule.
- *Autonomous*: The system selects one of all the applicable transformations in a non-deterministic way. This corresponds exactly to the notion of internal choices in process calculi. Accordingly, we may represent all this reductions by hiding

the actual name of the applied rule. That is, a rewriting step $G \Rightarrow_p G'$ in which p is an autonomous reconfiguration can be represented as $G \Rightarrow_\tau G'$, where τ stands for an autonomous change.

- *Pre-defined*: Pre-defined selection is an 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 of the form if - then - else - of process calculi). This can be mapped into graph grammars as the definition of priorities in the selection of productions to be applied. As shown in [10], application conditions can be used as priorities for restricting the order in which rules are applied.

We will differentiate among autonomous and external dynamism by considering whether the choice is explicit or autonomous. Let $\mathcal{G} = \langle T, G_{in}, P_{ext} \cup P_{self} \rangle$ be a grammar, where P_{ext} stands for the set of all reconfigurations that are controlled by the environment, while P_{self} contains all the autonomous productions. We say $\mathcal{G}_\mathcal{A}$ has (i) self dynamism if $P_{ext} = \emptyset$, (ii) external dynamism if $P_{self} = \emptyset$, or (iii) mixed dynamism otherwise.

Assuming that all rewriting steps $G \Rightarrow_p G'$ are written $G \Rightarrow_\tau G'$ when $p \in P_{self}$, we define the following sets associated to the grammar $\mathcal{G} = \langle T, G_{in}, P_{ext} \cup P_{self} \rangle$:

- The set $\mathcal{S}(\mathcal{G})$ of autonomous or self reconfigurations, i.e., the set of all configurations reachable by applying autonomous changes is: $\mathcal{S}(\mathcal{G}) = \{G \mid G_{in} \Rightarrow_{\tau^*} G\}$.
- The set $\mathcal{E}_c(\mathcal{G})$ of reconfigurations associated to an external sequence $c = p_1, \ldots, p_n$ of commands:
$\mathcal{E}_c(\mathcal{G}) = \{G \mid G_{in} \Rightarrow_{c'} G \ \wedge c' = \tau^*, p_1, \tau^*, \ldots, \tau^*, p_n, \tau^*\}$. Note $\mathcal{E}_c(\mathcal{G})$ 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.

Clearly, $\mathcal{S}(\mathcal{G})$ and $\mathcal{E}_c(\mathcal{G})$ are subsets of $\mathcal{R}(\mathcal{G})$. Hence, we can proceed as in Section 2, and formulate some verification problems. In particular, we can specialise the problem $\mathcal{R}(\mathcal{G}) \subseteq \mathcal{D}_\mathsf{P}(\mathcal{G})$ to either $\mathcal{S}(\mathcal{G}) \subseteq \mathcal{D}_\mathsf{P}(\mathcal{G})$ or $\mathcal{E}_c(\mathcal{G}) \subseteq \mathcal{D}_\mathsf{P}(\mathcal{G})$. The last relation is particular interesting when considering ad-hoc or constructible dynamism. In this case, it is possible to check whether a particular reconfiguration program may produce acceptable configurations.

5 Final Remarks

In this work we have characterised different aspects of dynamic reconfiguration as particular features of graph rewriting systems. By taking advantage of this framework, we have distill whether such kinds of dynamisms allow for posing typical questions about the completeness and correctness of the architectural specification. Figure 12 summarises the conclusions for the four selected types of dynamisms, i.e., *programmed*, *ad-hoc*, *constructible*, and *repairing*.

Dynamicity	Correctness	Completeness
Programmed	+	+
Ad hoc	-	-
Constructible	-/+	-/+
Programmed	+	+

Figure 12. Style of dynamicity vs completeness and correctness

As mention in Section 2, given a characterization of all desirable configurations of a programmable architecture, e.g., by defining a property P that should hold in every configuration, then it would be possible to prove whether the architectural specification is correct (by showing that P holds in every reachable configuration) and complete (by proving any configuration satisfying P is reachable). Nevertheless, such questions are meaningless for ad hoc dynamicity, since every configuration is potentially reachable when ad hoc dynamicity is considered. Analogous is the situation for constructible dynamism. Nevertheless, some kind of weak analysis could be performed in this case. For instance, to prove that particular configurations are not reachable when the reconfiguration language forbid some kind of programs. However, correctness and completeness properties could be associated to repairing dynamism. But, differently from programmable dynamism, some reachable configurations of a repairing architecture may be non desirable. Nevertheless, those configurations should be transformed into a desirable one by using repairing rules. The main idea is that undesirable configurations are characterized as those reachable configurations in which some repairing rule is applicable. Then, correctness and completeness properties involve those reachable configurations that are desirable.

Actually, the above characterization corresponds to the case in which transformations are all autonomous, i.e., when we assume self dynamism. Nevertheless, when external dynamism is considered, also correctness and completeness properties over ad hoc and constructible architectures can be formulated. For instance, given a particular (set of) desirable configuration(s) it can be proved whether a particular transformation or configuration program selected by a programmer produces a desirable configuration. Even more interesting is the case in which mixed dynamism is considered. Assume an ad hoc architecture where some productions are considered external and others autonomous or self. In this case, external transformations account for the reconfigurations activated by a user, while autonomous transformations model the actual program that performs the transformation (a kind of scripting). In this case, it would be possible to check whether a particular script produces a correct configuration when it is applied over a specific configuration.

In this paper we have identified classes of properties that can be naturally associated to any of such kinds of dynamicities. Next work will be approach the problem of verifying such properties over graph grammar specifications. In particular, we have in mind to use Alloy [13,14] for attempting this task and we are going to concentrate our work on proving properties associated to each kind of dynamic software architecture.

References

[1] R. Allen, R. Douence, and D. Garlan. Specifying and analyzing dynamic software architectures. In *Proceedings of FASE'98*, 1998.

[2] J. Andersson. Issues in dynamic software architectures. In *Proceedings of ISAW4*, pages 111–114, June 2000.

[3] L. Baresi, R. Heckel, S. Thöne, and D. Varró. Modeling and validation of service-oriented architectures: application vs. style. In *ESEC / SIGSOFT FSE*, pages 68–77, 2003.

[4] L. Baresi, R. Heckel, S. Thöne, and D. Varró. Style-based refinement of dynamic software architectures. In *WICSA*, pages 155–166, 2004.

[5] G. Berry and G. Boudol. The chemical abstract machine. *Theor. Comput. Sci.*, 96(1):217–248, 1992.

[6] J. S. Bradbury, J. R. Cordy, J. Dingel, and M. Wermelinger. A survey of self-management in dynamic software architecture specifications. In *WOSS*, pages 28–33, 2004.

[7] M. Endler. A language for implementing generic dynamic reconfigurations of distributed programs. In *In Proceedings of BSCN 94*, pages 175–187, 1994.

[8] D. Garlan and B. R. Schmerl. Model-based adaptation for self-healing systems. In *WOSS*, pages 27–32, 2002.

[9] I. Georgiadis, J. Magee, and J. Kramer. Self-organising software architectures for distributed systems. In *WOSS*, pages 33–38, 2002.

[10] A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Fundam. Inform.*, 26(3/4):287–313, 1996.

[11] D. Hirsch, P. Inverardi, and U. Montanari. Graph grammars and constraint solving for software architecture styles. In *Proceedings of ISAW '98*, pages 69–72. ACM Press, 1998.

[12] D. Hirsch, P. Inverardi, and U. Montanari. Reconfiguration of software architecture styles with name mobility. In *COORDINATION*, pages 148–163, 2000.

[13] D. Jackson. Alloy: a lightweight object modelling notation. In *ACM Transactions on Software Engineering and Methodology*, 2002.

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

[15] M. Löwe. Algebraic approach to single-pushout graph transformation. *Theor. Comput. Sci.*, 109(1&2):181–224, 1993.

[16] M. Löwe, M. Korff, and A. Waner. An algebraic framework for the transformation of attributed graphs. pages 185–199, 1993.

[17] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE TSE*, 26(1):70–93, 2000.

[18] D. L. Métayer. Describing software architecture styles using graph grammars. *IEEE Trans. Software Eng.*, 24(7):521–533, 1998.

[19] P. Oreizy. Issues in the runtime modification of software architecture. *Elettronics Letters*, (UCI-ICS-96-35), 1996.

[20] P. Oreizy, M. Gorlick, R. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, and A. Wolf. An architecture-based approach to self-adaptive software. In *Intelligent Systems, IEEE*, volume 14, pages 54–62, 1999.

[21] B. R. Schmerl and D. Garlan. Exploiting architectural design knowledge to support self-repairing systems. In *SEKE*, pages 241–248, 2002.

[22] M. Shaw and D. Garlan. Software architecture: Perspectives on an emerging discipline. In *Prentice Hall, NJ. USA*, 1996.

[23] M. Wermelinger. Towards a chemical model for software architecture reconfiguration. *IEE Proceedings - Software*, 145(5):130–136, 1998.