On Recovering from Run-time Misbehaviour in ADR[*]

Kyriakos Poyias Emilio Tuosto
Department of Computer Science, University of Leicester, UK
`kyriakos@le.ac.uk` `emilio@le.ac.uk`

We propose a monitoring mechanism for recording the evolution of systems after certain computations, maintaining the history in a tree-like structure. Technically, we develop the monitoring mechanism in a variant of ADR (after Architectural Design Rewriting), a rule-based formal framework for modelling the evolution of architectures of systems.

The hierarchical nature of ADR allows us to take full advantage of the tree-like structure of the monitoring mechanism. We exploit this mechanism to formally define new rewriting mechanisms for ADR reconfiguration rules. Also, by monitoring the evolution we propose a way of identifying which part of a system has been affected when unexpected run-time behaviours emerge. Moreover, we propose a methodology to suggest reconfigurations that could potentially lead the system in a non-erroneous state.

1 Introduction

We present a technical development of the *Architectural Design Rewriting* model (ADR) [4] that combines the features of ADR described in [4, 11]. We take the motivation of our work from the problems arising in modern software development. Software systems are no longer static and become more and more dynamic; because of their very interactive nature, such systems are starting to be studied under new angles [6]. For instance, software needs to adapt to the (often unpredictable) changes of the (virtual and physical) environment it operates in. The term *autonomic computing* has been coined to mark such systems [7], which present new degrees of complexity since they require high levels of flexibility and adaptiveness [8].

Such complexity calls for rigorous methods at very early stages of software development. Architectural Description Languages (ADLs) used to design such systems have to be able to guarantee software quality and correctness by being flexible to adapt from their initial designs, and also be able to predict the possible problems that could arise during the execution of such systems. Formal approaches aim to devise robust engineering practices to form reliable software products to mitigate the issues described above. Arguably, those approaches focus on software behaviour; correctness and efficiency of software play in fact a crucial role not only in critical systems but also in daily-life applications. In the design phase, semi-formal methods are typically adopted; as an example, the use of modelling languages is combined with design patterns to devise a model that can be checked. This approach may involve formal techniques (e.g., type or model checking) to guarantee properties of applications while non-formal techniques (or tools not supported by formal approaches) are typically used to tackle architectural design aspects. Our research agenda envisages the combination of those approaches with techniques to address the issues above at the design level. We believe that a rigorous treatment at the design level would allow to identify and solve many problems that are currently tackled only by inspecting or testing code.

We describe a formal framework that is able to tackle the architectural/structural aspects of the design and allow designers to identify and address problems at a higher level. In [11] we advocate a

[*]This work has been supported by FP7-PEOPLE-2011-IRSES MEALS

design-by-contract (DbC approach) for ADLs that allows the specification of *contracts* that architectures have to abide by. On these grounds, in [11] we propose a methodology that is able to compute structural "rearrangements" of systems' architecture to adapt themselves when an erroneous state is reached. Technically, the DbC approach in [11] is developed by extending ADR [4] with *asserted production rules*, that is rules for architectural transformations equipped with logical *pre-* and *post-conditions*. In this way, asserted productions become contracts that guarantee the *architectural style* when they are applied. The concept of architectural style is crucial in software architectures [12]. In ADR, the architectural style of a system is formalised in terms of *productions rules*, namely rules that can be used to generate specific configurations of the architectural elements. As surveyed in § 2, ADR models architectures as *(hyper)graphs* that is a set of *(hyper)edges* sharing some nodes; respectively, edges represent architectural elements (at some level of abstraction) while nodes represent components' interfaces. Also, ADR production rules take the form $p : L \to R$ where L is an edge and R an (hyper)graph; rule p is meant to replace L with R within a given graph. The main contribution in [11] is an algorithm that computes a weakest pre-condition ψ out of a post-condition φ and a production rule. We prove a theorem that guarantees that the application of the rule to a configuration satisfying ψ yields a configuration satisfying the post-condition φ. This algorithm can be used to compute a reconfiguration if the current configuration violates the invariant. Roughly, in [11] we envisage architectural styles according to the equation:

$$\text{architectural style} \quad = \quad \text{production rules} + \text{invariants} \tag{1}$$

where an invariant is the property the designer requires of the application.

A summary of our contributions. The main contributions of this paper can be summarised as an extension and a refinement of the methodology in [11].

The extension consists in the adaptation of the methodology to encompass *reconfiguration mechanism* of ADR. As a matter of fact, ADR features complex reconfigurations that cannot be captured by production rules. Such complex reconfigurations can be envisaged as a model of run-time evolution of systems that describe what complex rearrangements could happen during execution. This is technically done by specifying term rewriting rules in an algebra where terms are interpreted as proofs of the style of graphs. Here we broaden the applicability of the methodology in [11] to a more general setting that allows the iteration of the methodology in [11] when reconfigurations violate the style. A limitation of the methodology in [11] was due to the fact that style violations could be fixed only considering "top-down" application of productions. In this paper we take into account also violations of the style due to complex reconfigurations. To achieve this we have to identify the "positions" in the system where violations occur. This allows us to apply our methodology only to the parts of the system affected by the ill reconfiguration. Also, we propose here a systematic reiteration of the methodology when an immediate way to recover the style cannot be found. This yields a more general and efficient framework.

Intuitively, the equation (1) now becomes

$$\text{architectural style} \ = \ \text{production rules} + \text{reconfiguration rules} + \text{invariants}$$

This generalisation is possible due to the introduction of a monitoring approach that fully exploits the features of ADR.

The refinement we propose here regards the rewriting mechanism of ADR and, more importantly, its underlying monitoring capability. Indeed, as observed in [4], a distinctive aspect of ADR is that it features the canonical view of software architectures in terms of connected architectural elements as well as a hierarchical view of software architectures that is paramount in the design phase. Although [4] advocates the use of the architectural view as a useful mechanism to be exploited in complex reconfigurations, no actual formalisation has been provided on how this could be achieved and the parsing features of ADR had been only sketched in [4]. More precisely, we start by proposing minor changes to the original rewriting mechanism of ADR that distinguishes edge as terminal and non-terminal at the type level and allows only non-terminal edges to be rewritten (for each edge type, either all the edges of that type can be rewritten or none of them). Our generalisation eliminates such distinction; an edge can be rewritten if it is marked as "replaceable" in the graph. Therefore, we allow edges of the same type to be rewritten or not depending on how they are marked in the graph. (We note that the original ADR rewriting mechanism can still be obtained: if one decides that an edge type is non-terminal, then all edges of that type have to be replaceable while for types of terminal edges, all edges have to be marked as non-replaceable.) Besides some simplification in the technical presentation of ADR (which is now more uniform), such generalisation brings in extra flexibility. In fact, a replaceable edge can be refined by introducing new versions of a rule that differs only for the "replaceability" of some edges.

In addition we introduce a monitoring mechanism that is also exploited to define an efficient parsing of ADR graphs. Our monitoring mechanism keeps track of the application of reconfiguration rules and uses such information when the graph has to be parsed (to identify the part that violates a style). The application of a reconfiguration rule affects such information that need to be updated accordingly.

Summing up, one can enforce the architectural style of the system in presence of complex reconfiguration that may violate the style; this can be achieved by

1. defining a monitoring mechanism,
2. repeatedly adapting the methodology in [11] exploiting the parsing features defined here.

Structure of the paper. § 2 overviews ADR and introduces its new variant as well as it summarises the results in [11]. § 3 defines our monitoring approach. § 4 gives the new rewriting mechanism hinging on our monitoring approach. § 5 gives the refinement of our methodology. § 6 draws some conclusions.

2 A Variant of ADR

In the following, $\mathfrak{N}$ and $\mathfrak{E}$ are two countably infinite and disjoint sets (of nodes and edges respectively), $X^* \stackrel{\text{def}}{=} \{(x_1,\ldots,x_n) \mid x_1,\ldots,x_n \in X\}$ is the set of finite lists on a set X, and $\tilde{x}$ ranges over X^*. Also, abusing notation, we sometimes use $\tilde{x}$ to indicate its underlying set of elements.

Definition 1 ((Hyper)graphs and morphisms [4]) *A* (hyper)graph *is a tuple $G = \langle V, E, t\rangle$ where $V \subseteq \mathfrak{N}$ and $E \subseteq \mathfrak{E}$ are finite and $t : E \to V^*$ is the* tentacle function *connecting edges $e \in E$ to a list of nodes; the* arity *of e is the length of $t_G(e)$. It is convenient to write $e(\tilde{u}) \in G$ for $e \in E_G$, $t_G(e) = \tilde{u} \subseteq V_G$; also, given a graph G, V_G, E_G, and t_G respectively denote the nodes, the edges, and the tentacle function of G.*

Given two graphs G and H, a morphism from G to H *is a pair of functions $\langle \sigma_V : V_G \to V_H, \sigma_E : E_G \to E_H\rangle$ s.t. σ_V and σ_E preserve the tentacle functions, i.e. $\sigma_V^* \circ t_G = t_H \circ \sigma_E$, where σ_V^* is the homomorphic extension of σ_V to V_G^*.*

In ADR, graphs are typed over a fixed type graph via typing morphisms. As usual an ADR graph G *is typed over a type graph* Γ *through* τ_G if τ_G is a morphism from G to Γ.

Definition 2 (Typed graphs) *Let Γ be a type graph. An ADR graph G* is a (hyper)graph typed over Γ through τ_G *if τ_G is a morphism from G to Γ.*

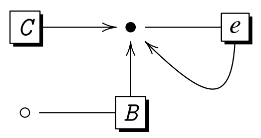
Example 1 *Take the type graph $\Gamma = \langle V, E, t\rangle$ where $V = \{\bullet, \circ\} \subseteq \mathfrak{N}$, $E = \{C, B, FF, Fls, Fl, BF, P, PF\} \subseteq \mathfrak{E}$, and $t : C \mapsto (\bullet)$, $t : B \mapsto (\bullet, \circ)$, and $t : e \mapsto (\bullet, \bullet)$ for each $e \in E \setminus \{B, C\}$.*
The graph $G = \langle \{u_1, u_2, u_3, u_4\}, \{ff, fl_1, fl_2\}, t'\rangle$ where t' is defined as $t' : ff \mapsto (u_2, u_1)$, $t' : fl_1 \mapsto (u_3, u_2)$, and $t' : fl_2 \mapsto (u_4, u_2)$ can be typed on Γ by τ_G mapping all the nodes to $\bullet$, fl_1 and fl_2 to Fls, and ff to FF.

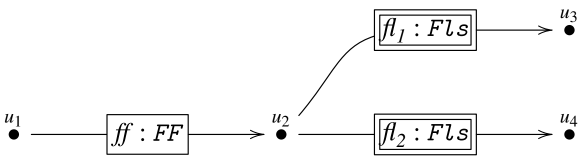
Hereafter, we fix a typed graph Γ and tacitly assume that all graphs G are typed over Γ via a morphism τ_G. Intuitively, Γ yields the *vocabulary* of the architectural elements to be used in the designs; moreover, Γ specifies how these elements can be connected together (e.g., as in Example 1).

For technical reasons we introduce a slight variant of ADR; instead of considering edges as *non-terminal* and *terminal* edges, the new version of ADR allow more liberal rewriting mechanism by marking in a graph the edges that can be replaced. Technically, this is obtained by considering pairs $\langle G, \theta\rangle$ where G is a graph and $\theta : E_G \to \{0, 1\}$ is the *replaceability map*; an edge $e \in E_G$ is *replaceable* iff $\theta(e) = 1$. Abusing notation we will implicitly assume that any graph G is equipped with a replaceablility map which we will denote by θ_G.

Type and typed graphs have a convenient visual notation. Nodes are circles and edges are drawn as (labelled) boxes; tentacles are depicted as lines connecting boxes to circles; conventionally, directed tentacles indicate the first node attached to the edge and the others are taken clockwise. The boxes of edges of type graphs are shaded, while the edges in a graph are either single- and double-lined boxes; the former represent non-replaceable edges while the latter represent replaceable ones. The visual notation for typed graphs include the graph and its typing morphism. Nodes are paired with their types while an edge label $e : e'$ represents the fact that the typing morphism maps the edge e of the graph to the edge e' of the type graph.

Example 2 *In the visual notation described above, the type graph Γ and the graph G of Example 1 can be respectively drawn as*





where, to simplify the type graph, we use $e \in E \setminus \{B, C\}$ (instead of drawing an edge for each edge of Γ with arity two.

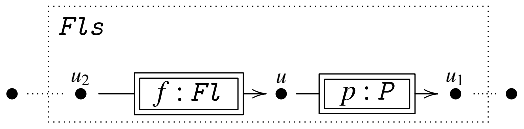
Definition 3 (Typed Graph morphisms) *A* morphism *between Γ-typed graphs $f : G_1 \to G_2$ is a* typed graph morphism *if it preserves the typing, i.e. such that $\tau_{G_1} = \tau_{G_2} \circ f$.*

Note that replaceablility maps are not considered in Definition 2.

Definition 4 (Productions) *A* (design) production *p is a tuple $\langle L, R, i : V_L \to V_R\rangle$ where L is a graph consisting of a single repleaceble edge attached to distinct nodes and R is a graph; the nodes in $Im(i)$ (the image of i) are called* interface nodes.

Design productions can be thought of as rewriting rules that, when applied to a graph G, replace a replaceable (hyper)edge of G matching L with a fresh copy of R (we remark that our morphisms are type-preserving). Also productions have a suitable visual representation illustrated in the next example.

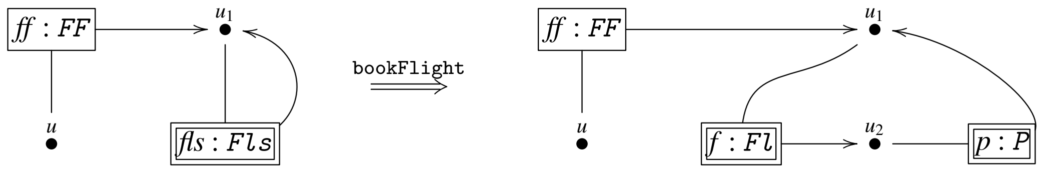
Example 3 *The graphical representation below represents a design production.*



Since the production above will be used later (cf. Example 9) we will refer to it as `bookFlight`. *The left-hand-side (LHS) of* `bookFlight` *is an edge of type* `Fls` *(denoted in the left-upper corner of the dotted-box) whose nodes are those outside the dotted box; we omit the identities of such nodes when immaterial. The right-hand-side (RHS) of* `bookFlight` *is the graph inside the dotted box. The mapping i of* `bookFlight` *is represented by the dotted lines.*

The next example illustrates how productions are applied to graphs; the details will be given in § 2 for *asserted productions*, which encompass ADR productions.

Example 4 *Consider the production* `bookFlight` *of Example 3. Below, the unique edge of type* `Fls` *is replaced by an instance of the RHS of* `bookFlight`.



Note that the rest of the graph (consisting only of the edge ff) including the interface nodes is left unchanged while a fresh node u_2 is created.

We overview the Design by Contract (DbC) approach for ADR introduced in [11]. Note that the variant of ADR given in § 2 generalises the rewriting mechanism originally defined in [4], therefore the results in [11] can be easily adapted to the variant of ADR presented here.

Properties of graphs are expressed in a simple logic tailored for ADR. In the following we let $D, D', \dots$ range over edges of Γ.

Definition 5 (ADR logic [11]) *Let* V *be a countably infinite set of variables for nodes (ranged over by* $\mathsf{x}, \mathsf{y}, \mathsf{z}, \dots$*). The set* $\mathscr{L}$ *of* (graph) formulae *for ADR is given by the following grammar:*

$$\psi, \varphi \quad ::= \quad \mathsf{x} = \mathsf{y} \quad | \quad \top \quad | \quad \neg\varphi \quad | \quad \varphi_1 \wedge \varphi_2 \quad | \quad \forall D(\tilde{\mathsf{x}}).\varphi$$

In formulae of the form $\forall D(\tilde{\mathsf{x}}).\varphi$, the occurrences of $\mathsf{y} \in \tilde{\mathsf{x}}$ in φ are bound, *$\tilde{\mathsf{x}}$ has the length of the arity of D and $\tilde{\mathsf{x}}$ are pairwise distinct.*

Basically, $\mathscr{L}$ is a propositional logic to predicate on (in)equalities of nodes and it is parametrised with respect to the type graph Γ used in quantification. Variables not in the scope of a quantifier are free and the set $\mathrm{fv}(\varphi)$ of *free variables* of $\varphi \in \mathscr{L}$ is defined accordingly. The models of $\mathscr{L}$ are ADR graphs together with an interpretation of the free variables of formulae.

Definition 6 (Satisfaction relation) *A graph G* satisfies $\varphi \in \mathscr{L}$ under the assignment $h : \mathsf{V} \to V_G$ *(in symbols $G \models_h \varphi$) iff*

$$\begin{array}{llll}
\varphi \equiv \top, & & & \textit{or} \\
\varphi \equiv \mathsf{x} = \mathsf{y} & \textit{and} & h(\mathsf{x}) = h(\mathsf{y}), & \textit{or} \\
\varphi \equiv \neg\varphi' & \textit{and} & G \not\models_h \varphi', & \textit{or} \\
\varphi \equiv \varphi_1 \wedge \varphi_2 & \textit{and} & G \models_h \varphi_1 \textit{ and } G \models_h \varphi_2, & \textit{or} \\
\varphi \equiv \forall D(\tilde{\mathsf{x}}).\varphi & \textit{and} & G \models_{h[\tilde{\mathsf{x}} \mapsto \tilde{u}]} \varphi \textit{ for any } d(\tilde{u}) \in G & \textit{s.t. } \tau_G(d) = D
\end{array}$$

Note that in the last clause of Definition 6, each bound variable in $\tilde{\mathsf{x}}$ is instantiated with a node. It is easy to prove that we can restrict to finite mappings that only assign the free variables of formulae. Namely, for each $h, h' : \mathsf{V} \to V_G$, if $h|_{\mathrm{fv}(\varphi)} = h'|_{\mathrm{fv}(\varphi)}$ then $G \models_h \varphi$ iff $G \models_{h'} \varphi$. We write $G \models \varphi$ when $\mathrm{fv}(\varphi) = \emptyset$.

$$\begin{tikzcd}
\psi \arrow[r, "h"] \arrow[ddr, equal] & L \arrow[r, "i"] \arrow[dd, "\sigma"] & R \arrow[d, equal, "\iota"] & \varphi \arrow[l, "h'"'] \\
 & & R' \arrow[d, hook] & \\
 & G \arrow[r, "\pi"] & G' &
\end{tikzcd}$$

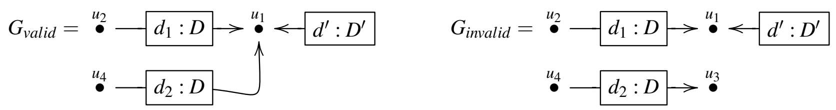
Figure 1: Asserted design productions

Example 5 *Consider the formulae*

$$\texttt{noEdge}\langle D\rangle \overset{\text{def}}{=} \forall D(\tilde{\mathsf{x}}).\bot \tag{2}$$

$$\phi_{ex} \overset{\text{def}}{=} \forall D(\mathsf{x},\mathsf{y}).\exists D'(\mathsf{z}).\mathsf{x} = \mathsf{z} \tag{3}$$

Formula (2) *characterises the graphs that do not contain edges of a given type while the formula* (3) *describes graphs such that each edge of type D is connected to one of type D′ on the first tentacle. For instance, consider the graphs*



then G_{valid} satisfies ϕ_{ex} whereas $G_{invalid}$ does not, because d_2 is not connected to any edge of type D'.

Fix an ADR production $p = \langle L, R, i\rangle$. Our notion of contracts hinges on *asserted productions*, namely ADR productions decorated with pre- and post-conditions expressed in the logic $\mathscr{L}$. Given $\psi, \varphi \in \mathscr{L}$ and two assignments $h, h' : \mathsf{V} \to \mathfrak{N}$, an *asserted production* is an expression of the form

$$\{\psi, h\}\ p\ \{\varphi, h'\} \qquad \text{where} \qquad h(\text{fv}(\psi)) \subseteq V_L \text{ and } h'(\text{fv}(\varphi)) \subseteq V_R \tag{4}$$

An asserted production generalises ADR productions and it intuitively requires that if p is applied to a graph G that satisfies ψ then the resulting graph is expected to satisfy φ. The assignments h and h' in (4) allow pre- and post-conditions to predicate on nodes occurring in the LHS or the RHS of p.

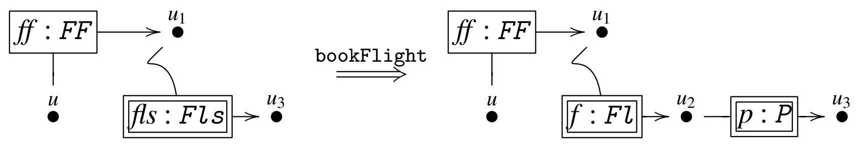
Operationally, an asserted production π can be applied to a graph G by replacing an "instance" of the LHS in G (identified by an *matching homomorphism*) with a new instance of the RHS and connecting the interface nodes accordingly, provided that G satisfies the precondition of π (under the matching homomorphism). For the variant of ADR proposed in § 2 we just have to impose the condition that the homomorphic image of the LHS of π has to be a replaceable edge. This is schematically illustrated in Figure 1 (cf. [11]) and demonstrated in Examples 6 and 7.

Example 6 *Consider the production* `bookFlight` *given in Example 3 and the asserted production*

$$\pi \overset{\text{def}}{=} \{\psi, \emptyset\}\ \texttt{bookFlight}\ \{\top, \emptyset\} \qquad \textit{where} \qquad \psi \overset{\text{def}}{=} \forall \mathit{Fls}(\mathsf{x},\mathsf{y}).\mathsf{x} \neq \mathsf{y}$$

Then, π cannot be applied to the leftmost graph G in the rewriting of Example 4 because $G \not\models \psi$ (under the unique morphism σ from L to G). In fact, x *and* y *are mapped to the same node u_1 of G.*

Example 7 *The rewriting below is obtained by applying π in Example 6.*



The edge fls is replaced by an isomorphic instance of R preserving the interface nodes u_1 and u_3.

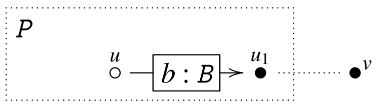
Note that the application of an asserted production generalises the hyper-edge replacement mechanism of ADR; in fact, $\{\top,\emptyset\}\ p\ \{\top,\emptyset\}$ applies exactly as normal ADR productions.

An asserted production π is *valid* when any application of π to a graph satisfying the precondition of π yields a graph satisfying the post condition of π. Obviously, not all asserted productions $\{\psi,h\}\ p\ \{\varphi,h'\}$ are valid (this can be trivially noted by taking φ to be $\bot$). In [11] we define an algorithm[1] $\mathscr{W}_{h'}(p,\varphi)$ that, given a production p and a post-condition φ, returns a weakest pre-condition ψ so that $\{\psi,h\}\ p\ \{\varphi,h'\}$ is valid.

The next example is adapted from [11].

Example 8 *Consider $\varphi \in \mathscr{L}$ and the production* `pay` *below:*

$\varphi \overset{\text{def}}{=} \forall B(\mathsf{x},\mathsf{y}).\forall C(\mathsf{z}).\mathsf{y} = \mathsf{z}$ $\qquad \mathtt{pay} \overset{\text{def}}{=}$



We remark that the post-condition φ requires that every edge of type B is connected (on its second tentacle) to every edge of type C. The computed weakest pre-condition is

$$\mathscr{W}_{\emptyset}(\mathtt{pay},\varphi) \;=\; \mathtt{noEdge}\langle C\rangle \;\wedge\; \forall B(\mathsf{x},\mathsf{y}).\forall C(\mathsf{z}).\mathtt{noEdge}\langle C\rangle \;\wedge\; \forall B(\mathsf{x},\mathsf{y}).\forall C(\mathsf{z}).\mathsf{y} = \mathsf{z}$$

We remark that $\mathscr{W}_{\emptyset}(\mathtt{pay},\varphi)$ imposes that for the validity of the asserted production it is necessary that the graph does not have any edges of type C. In fact, production `pay` *will generate an edge of type B whose second tentacle is attached to an internal node u that cannot be shared with any edge of type C already appearing in the graph.*

3 Tracking ADR Architectural Reconfigurations

A key aspect of ADR is to envisage systems as ensembles of *designs*, that is components *with interfaces*. Designs are supposed to be generated by means of productions and can be subject to run-time reconfigurations modelled as *reconfiguration rules*. The use of productions yields two pivotal ingredients of ADR. For clarity, we consider only productions as it is just a matter of technicality to adapt this section to asserted productions.

Firstly, productions implicitly equip designs with a hierarchical structure that can be formalised as the “derivation tree” determining the design. In fact, a set of ADR productions induces a multi-sorted algebraic signature Σ where the sorts are the type edges in the type graph[2] and the operations are the productions themselves, once a total order on the edges in the RHS of the production is fixed. Hereafter,

[1] For simplicity, we ignore the assignments and environments that the algorithm in [11] uses to compute weakest preconditions.

[2] In the original ADR presentation, the sorts are just the non-terminal edges. In our variant, this can be simplified by taking all edges of the type graph as sorts.

we fix such an order[3] and, given the RHS R of a production, we write $R[j]$ for the j-th edge in R. With this construction, an ADR production becomes an operation with type

$$E_1 \times \ldots \times E_n \to L \tag{5}$$

where E_k is the type of the k-th edge in the RHS of the production (according to the chosen order on edges in the RHS) and L is the type of the edge in the LHS. In other words, an ADR production like in (5) can be envisaged as an operation in some *algebras of designs* that builds a design G of type L out of designs G_k of type E_k (for $1 \le k \le n$). This corresponds to a "bottom-up" development (whereby designs are assembled out of other components) and, as observed in [4], it parallels the "top-down" generation of designs (similar to context-free grammars) reviewed in § 2. Moreover, one could consider the terms (with sorted variables to model partial designs) built on Σ and adopt the obvious operational interpretation: G is obtained by replacing the j-th edge in the RHS with G_j (and connecting the interface nodes as appropriate). The elements of such term algebra correspond to the proof that a given design can be assigned some type.

Example 9 *The production* `bookFlight` *in Example 3 yields the operation*

$$\texttt{bookFlight} : \mathit{Fl} \times P \to \mathit{Fls}$$

assuming that in the chosen order, f is smaller than p.

Secondly, ADR exploits the algebraic view of productions to model complex architectural *reconfigurations* that cannot be captured by productions. In fact, design can evolve for instance when components have to be removed, added, or assembled in a different way. Architectural reconfigurations are naturally modelled as transformation of elements in the Σ-term algebra with variables $\texttt{Term}_{\Sigma,\mathscr{X}}$ (where $\mathscr{X}$ is the set of variables). Formally, this is achieved by defining a term rewriting system on $\texttt{Term}_{\Sigma,\mathscr{X}}$; namely, a reconfiguration rule takes the form

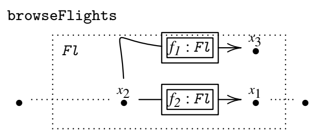
$$t \to t' \tag{6}$$

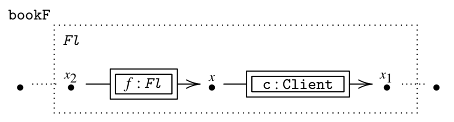
where $t, t' \in \texttt{Term}_{\Sigma,\mathscr{X}}$ are linear terms (that is each variable occurs at most once in t and similarly for t') and the variables occurring in t' also occur in t.

Example 10 *Combining the operation in Example 9 with the one associated to the production in Example 3 one could build the term* $\texttt{bookFlight}(x, \texttt{pay}(y))$ *of type* Fls *(provided that x is of type* Fl *and y is of type B.*

Below we give an example of simple reconfigurations.

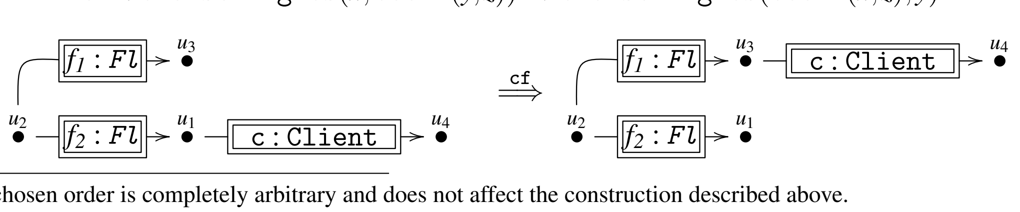
Example 11 *Consider the following productions:*





We can define the following reconfiguration rule:

$$\texttt{cf} : \texttt{browseFlights}(x, \texttt{bookF}(y,z)) \to \texttt{browseFlights}(\texttt{bookF}(x,z), y)$$



[3]The chosen order is completely arbitrary and does not affect the construction described above.

Observe that, unlike in the application of ADR productions, the identity of edge `c` is preserved when applying the reconfiguration rule `cf`. Also, for simplicity in Example11 we take `c` to be just a single edge, but the effect of `cf` would be the same if instead of edge `c` we had a complex graph of type `Client`: the whole graph would have been moved from node u_1 to node u_4.

A result in [4] shows that the simple condition that an ADR rewriting system where all reconfiguration rules of form (6) have t and t' of the same sort guarantees that the architectural style is preserved when the system evolves.

In this paper we exploit the algebraic presentation of ADR production and reconfiguration mechanisms and combine them together with a tracking mechanism that is used to recover possible run-time misbehaviour.

Def. 7 below formalises our tracking mechanism using some trees to record graphs' evolution due to productions and reconfigurations respectively. We introduce some technical machinery to formalise Def. 7. We consider forests of trees with nodes drawn from a set N; if $f : X \to Y$ is a partial map then we write $f(x)\uparrow$ when f is undefined on x and we let dom $f = X \setminus \{x \in X \mid f(x)\uparrow\}$. Hereafter, we fix a finite set of asserted productions $\mathscr{P}$ and $\mathscr{M}$ to denote the space of graph morphisms. A *tracking Environments* $\mathscr{T}$ is the product of two injective finite partial maps

$$\mathscr{T}^{(1)} : \mathsf{N} \to \mathfrak{E} \times \mathfrak{N}^*, \qquad \text{and} \qquad \mathscr{T}^{(2)} : \mathsf{N} \to \mathscr{P},$$

and we use $\mathbf{0}$ to denote the empty environment (that is the environment undefined on all $\mathsf{n} \in \mathsf{N}$).

Basically, given a forest T, we use an environment $\mathscr{T}$ (such that dom $\mathscr{T}$ is the set of nodes of T) so to decorate each node of T with two attributes:

- $\mathscr{T}^{(1)}(\mathsf{n})$ assigns an edge with its list of nodes to the nodes of T, and
- $\mathscr{T}^{(2)}(\mathsf{n})$ assigns a production to the node n in T.

It is convenient to write $\mathscr{T}(\mathsf{n}) \overset{as}{=} e(\tilde{x}) \cdot p$ when $\mathscr{T}^{(1)}(\mathsf{n}) = e(\tilde{x})$ and $\mathscr{T}^{(2)}(\mathsf{n}) = p$. Also, in the following we use a notation inspired by object-oriented programming to manipulate trees; more precisely, we consider trees T (and their nodes n) as objects and write $T.\mathsf{addTree}(\mathsf{n}, T_1, \ldots, T_k)$ to add the trees T_h as sub-trees of T by rooting them at the node n in T; that is, the resulting tree will be T where node n has the root of $T_1, \ldots, T_k$ as new children. Also, we let deg n to be the degree of a node n, $\mathsf{n}[j]$ to be its j-th child, and (abusing notation) we allow ourselves to identify trees consisting only of a root with the root node.

Definition 7 (Tracking productions) *Let $G_0, \ldots, G_m$ be a sequence of graphs s.t. $0 \le j < m$, G_{j+1} is obtained from G_j by applying a production $p_j \in \mathscr{P}$ with morphisms $\sigma'_j : L_j \to G_j$ and $\sigma_j : R_j \to G_{j+1}$ where L_j and R_j are the LHS and RHS of p_j, respectively.*

We associate to each G_j a forest of tracking trees T_j *and a* tracking environment $\mathscr{T}_j$ *as follows:*

- *let r be the number of edges in G_0, forest $T_0 = \mathsf{n}_1, \ldots, \mathsf{n}_r$ consists of r single-node trees with roots $\mathsf{n}_1, \ldots, \mathsf{n}_r$ taken pairwise distinct;*
- *environment $\mathscr{T}_0$ maps the m-th node in the forest T_0 to the m-th edge of G_0; formally, $\mathscr{T}_0[\mathsf{n}_m \mapsto e_m(\tilde{x}_m) \cdot \uparrow \mid 1 \le m \le r]$ where $e_m = G_0[m]$ and $\tilde{x}_m$ are the nodes e_m is attached to;*
- *assuming that k_j is the number of edges in the RHS of p_j (that is, k_j is the cardinality of E_{R_j}), let*

$$\mathsf{n} \textit{ be } \begin{cases} \mathsf{n}', & \textit{if the inverse image of } \sigma'_j(e_j(\tilde{x}_j)) \textit{ through } \mathscr{T}_j^{(1)} \textit{ is } \{\mathsf{n}'\} \\ \textit{a fresh node} & \textit{otherwise} \end{cases}$$

and, for $1 \le l \le k_j$, let T'_l be a tree made of just a fresh root node, then

$$T_{j+1} = T_j.\mathsf{addTree}(\mathsf{n}, T'_1, \ldots, T'_{k_j})$$

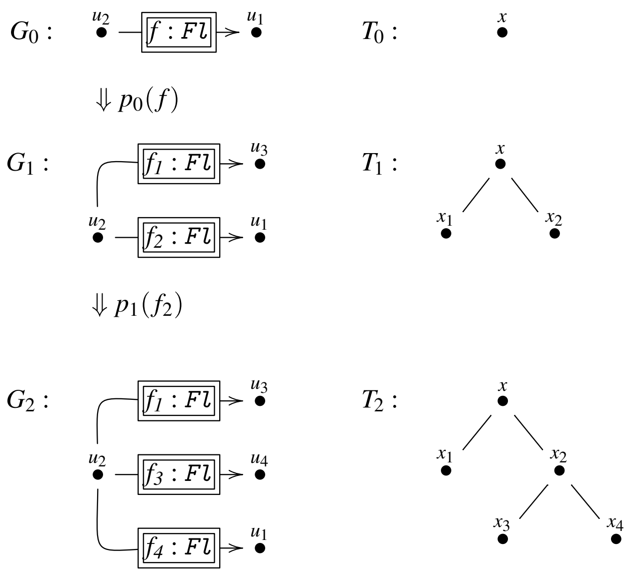
where the inverse image of $\sigma'_j(e_j(\tilde{x}_j))$ *through* $\mathscr{T}_j^{(1)}$ *is the singleton* $\{\mathsf{n}\}$.

- *environment* $\mathscr{T}_{j+1}$ *is obtained by updating* $\mathscr{T}_j$ *in the following way:*

$$\mathscr{T}_{j+1} = \begin{cases} \mathscr{T}_j[\mathsf{n} \mapsto \sigma'_j(e_j(\tilde{x}_j)) \cdot p_j,\ \mathsf{T}'_l \mapsto \sigma_j(R_j[l]) \cdot \uparrow \ \big|\ l = 1, \ldots, k] & \text{if } \ E_{R_j} \neq \emptyset \\ \mathscr{T}_j[\mathsf{n} \mapsto \sigma'_j(e_j(\tilde{x}_j)) \cdot p_j,\ T'_1 \mapsto \uparrow \cdot \uparrow] & \text{if } \ E_{R_j} = \emptyset \end{cases}$$

Despite some technical intricacy, Def. 7 is conceptually simple. Basically, we add to T_j as many fresh nodes as the edges in the RHS of the production p_j; such nodes become the children of the node n in T_j associated with the LHS $\sigma'_j(e_j)$. Accordingly, the environment $\mathscr{T}_{j+1}$ updates $\mathscr{T}_j$ recording edges, productions, and morphisms associated to n and the fresh roots of T'_l. Observe that each forest T_j has r trees, with r the number of edges of G_0. Indeed, the evolution of G_0 involves only the replacement of such edges (and those produced by such replacements). Therefore, we can record the application of p_j to G_j in a node of one of the trees representing the evolution of one of the initial edges of G_0.

Example 12 *Consider the production* `browseFlights` *from Example 11. For presentation purposes we use* $p_j(e)$ *to indicate the* j*-th application of the production* `browseFlights` *on an edge* e.



$$\mathscr{T}_O : \quad \mathsf{x} \mapsto [f(u_1,u_2), \uparrow]$$

$$\mathscr{T}_1 : \quad \begin{array}{l} \mathsf{x} \mapsto [f(u_1,u_2), p_0] \\ \mathsf{x}_1 \mapsto [f_1(u_3,u_2), \uparrow] \\ \mathsf{x}_2 \mapsto [f_2(u_1,u_2), \uparrow] \end{array}$$

$$\mathscr{T}_2 : \quad \begin{array}{l} \mathsf{x} \mapsto [f(u_1,u_2), p_0] \\ \mathsf{x}_1 \mapsto [f_1(u_3,u_2), \uparrow] \\ \mathsf{x}_2 \mapsto [f_2(u_1,u_2), p_1] \\ \mathsf{x}_3 \mapsto [f_3(u_4,u_2), \uparrow] \\ \mathsf{x}_4 \mapsto [f_4(u_1,u_2), \uparrow] \end{array}$$

Graph G_0 *represents the initial graph,* T_0 *its corresponding tree mapped to an initial environment* $\mathscr{T}_0$ *as defined in Def. 7. By applying* `browseFlights` *to* f *we obtain* G_1*. Since the RHS of* `browseFlights` *generates two edges we have to add two nodes in* T_0 *as children of the node corresponding to* f *to obtain* T_1*. Finally, we update the map of* x *in* $\mathscr{T}_0$ *to add the production applied to* f *and add two new mappings for the fresh nodes of* T_1 *to get* $\mathscr{T}_1$*. We now repeat this procedure for every application of a production.*

4 A new rewriting mechanism

Following the recording mechanism in Def. 7 we will suggest a new rewriting mechanism for ADR reconfigurations. Our approach hinges on tracking trees similar to those in Def. 7. The new rewriting approach will also allow us to keep track of changes due to reconfigurations.

Given a term $t \in \mathtt{Term}_{\Sigma,\mathscr{X}}$ and a tracking tree T, the relation $t \bowtie T$ holds iff t and T are isomorphic "up to the leaves" of t; more precisely, the tree obtained by considering just the internal nodes of t is isomorphic to T. This can be formalised as

$$t \bowtie T \quad \Longleftrightarrow \quad t \in \mathscr{X} \quad \text{or} \quad t = p(t_1,\dots,t_k) \wedge \mathscr{T}^2(\diamond) = p \wedge \bigwedge_{j=1,\dots,k} t_j \bowtie \diamond[j]$$

where $\diamond$ is the root of T; if x is a variable of t we also define

$$t^x \sqsubset T = \begin{cases} T & \text{if } t = x \\ t_j^x \sqsubset \diamond[j] & \text{if } t = p(t_1,\dots,t_k) \wedge \mathscr{T}^{(2)}(\diamond) = p \wedge \exists 1 \le j \le k : x \in t_j \\ \uparrow & \text{otherwise} \end{cases}$$

that returns the subtree corresponding to x if T can mach a path from the root of t to x.

Given a reconfiguration rule $\rho : t \to t'$, the relation $\_ \bowtie \_$ allows us to identify which parts of a graph matches the LHS of ρ exploiting the correspondence between tracking trees and graphs. A sub-tree T' of T *matches* t iff $t \bowtie T'$. Also, assuming $t \bowtie T'$ and given a variable x occurring in t, let T_x be the sub-tree of T' corresponding to x (obtained by applying $t^x \sqsubset T'$).

Definition 8 (From Term to Graph) *Let $p = \langle L, R, i : V_L \to V_R \rangle \in \mathscr{P}$.*

$$\gamma(t) = \begin{cases} (e,\ [n_1,\dots,n_h],\ \eta : x \mapsto e) & \text{if } t = x,\ [n_1,\dots,n_h] \text{ are fresh pairwise distinct nodes} \\ ((G_1 \cup \dots \cup G_r)\sigma,\ \delta,\ \eta_1;\sigma \cup \dots \cup \eta_r;\sigma) & \text{if } t = p(t_1,\dots,t_r),\quad \gamma(t_j) = (G_j, \delta_j, \eta_j) \text{ for } 1 \le j \le r \end{cases}$$

where, assuming $\delta_j = [\check{n}_1,\dots,\check{n}_{l_j}]$ for each $1 \le j \le r$ and fixed two graphs L' and R' isomorphic to L and R respectively such that all their nodes and edges are fresh with $[\hat{n}_1,\dots,\hat{n}_k]$ the nodes of L', and $\iota_L : L \to L'$ and $\iota_R : R \to R'$ the isomorphisms, then

- $\delta = \iota_R(i(\iota_L^{-1}([\hat{n}_1,\dots,\hat{n}_k])))$ *and*
- *for $1 \le j \le r$ and $1 \le m \le l_j$, $\sigma : \check{n}_m \mapsto \iota_R(\bar{n}_{j,m})$ where $[\bar{n}_{j,1},\dots,\bar{n}_{j,l_j}]$ are the nodes of the j-th edge of R.*

Def. 8 builds a graph $\gamma(t)$ out of a term t. Intuitively, γ inspects t "bottom-up" and it associates disjoint designs (graphs G_j with interfaces δ_j) to each subterm of t (note that fresh edges attached to fresh nodes are associated to each variable of t); then γ composes the disjoint designs according to the production p which is rendered by replacing the nodes through the substitution σ in Def. 8.

Def. 9 below establishes how to apply a reconfiguration rule to a graph.

Definition 9 (Applying reconfiguration rules) *Fix a reconfiguration rule $\rho : t \to t'$ with X being the set of variables of t, a graph G and a tracking tree of G, say T; let T' be a sub-tree of T such that $t \bowtie T'$. For $x \in X$, let $T'_x = t^x \sqsubset T'$ be the sub-tree of T' corresponding to x. An* application of ρ to G wrt T *is a graph $G' = G[G_L \mapsto G_t]$ where*

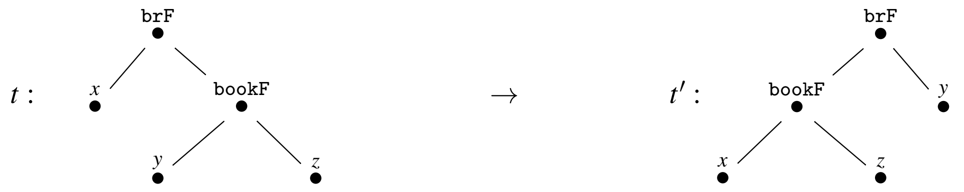
- $G_L = \bigcup_{l=1}^{n} \mathscr{T}^{(1)}(\mathsf{n}_l)$ *where $\mathsf{n}_1,\dots,\mathsf{n}_n$ are the leaves of T', and*
- $G_t = \gamma(t')[\eta(x) \mapsto G_x \mid x \in X]$ *where G_x is the sub-graph of G corresponding to variable x and it is defined as $G_x = \bigcup_{m=1}^{h_j} \mathscr{T}^{(1)}(\mathsf{n}_m)$ with $\mathsf{n}_{j,1},\dots,\mathsf{n}_{j,h_j}$ being the leaves of T'_x.*

Finally, we can replace the tree T' in T by introducing a new sub-tree T'' that corresponds to t' after we replace the variables of t' with their corresponding sub-trees T'_x. $\mathscr{T}$ now, maps all the nodes of T'' up to the sub-trees T'_x to the productions associated to them through t'.

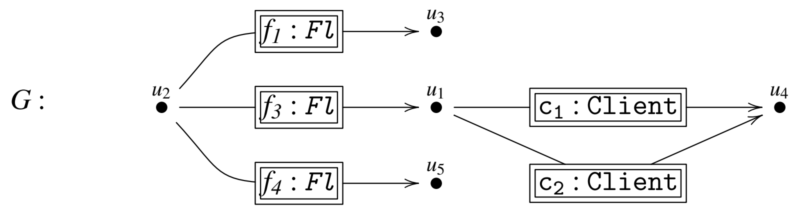
We observe that, using Def. 8, Def. 9 simply replaces the graph corresponding to $\gamma(t)$ with the graph corresponding to $\gamma(t')$ where the edges corresponding to the variables of t' are replaced by the corresponding subgraphs of G identified through the proper morphisms.

Example 13 *Let us consider again the productions* `cf` *in Example 11; for readability here we use t and t' to refer to the LHS and RHS of* `cf` *respectively and we also abbreviate* `browseFlights` *with* `brF`*:*

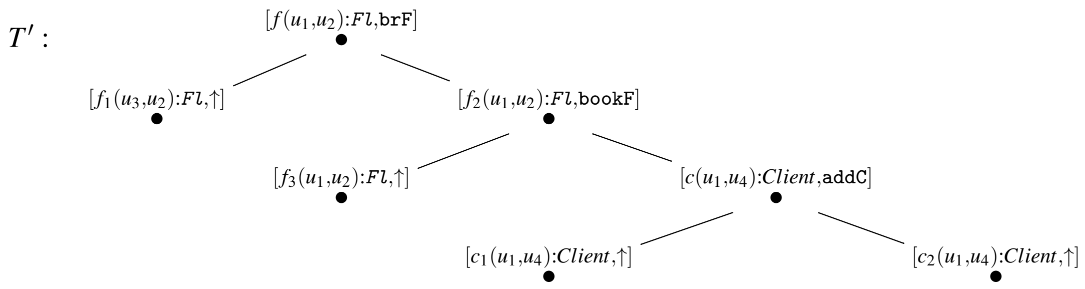
$$\texttt{cf} \quad : \quad \texttt{brF}(x, \texttt{bookF}(y,z)) \qquad \rightarrow \qquad \texttt{brF}(\texttt{bookF}(x,z),y)$$



Fix an environment $\mathscr{T}$ and consider the graph G below:



To apply `cf` *to G we have to identify a tree in the forest tracking G; this is done inspecting each tree of the forest with $\_ \bowtie \_$. Suppose that this yields the tree*

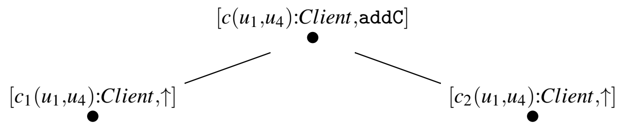


such that $t \bowtie T'$, then

1. *using $\_ \sqsubset T'$ we can obtain the sub-tree of T' corresponding to each variable of t and get*

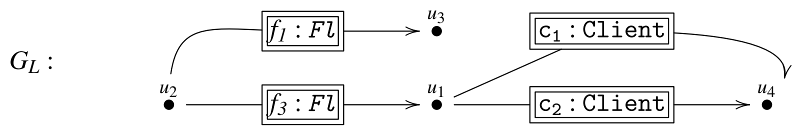
$$t^x \sqsubset T' = \overset{[f_1(u_3,u_2):Fl,\uparrow]}{\bullet} \qquad t^y \sqsubset T' = \overset{[f_3(u_1,u_2):Fl,\uparrow]}{\bullet}$$

$t^z \sqsubset T' =$

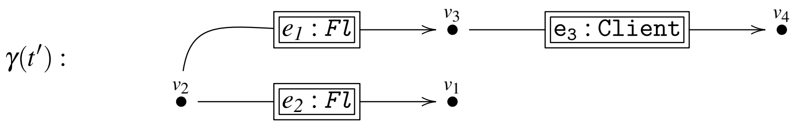


By taking the union of all the edges mapped through $\mathscr{T}$ to the leaves to the sub-trees corresponding to each variable we obtain sub-graphs of G corresponding to each one. Observe that z is not mapped to a single node T'. This is the reason we need to take the union of all the edges corresponding to the leaves of $t^z \sqsubset T'$ in order to obtain its sub-graph.

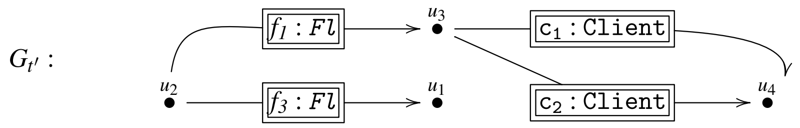
2. *identify the sub-graph of G corresponding to T'; similarly to step 1 we take the union of all the edges mapped through $\mathscr{T}$ to the leaves of T' to obtain:*



3. *use $\gamma(t')$ to construct a graph corresponding to t'*



4. *$\gamma(t')$ represents the graph corresponding to t' where in the place of the variables it contains a dummy edge of the appropriate type. This is where step 1 comes in place. We replace all the dummy edges in $\gamma(t')$ with the graphs corresponding to the variables of the dummy edges to obtain $G_{t'}$.*

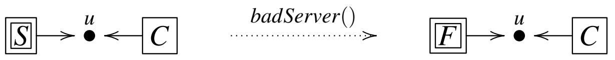


5. *the last step requires that we replace G_L in G with $G_{t'}$.*

5 Recovering invalid configurations

In [11] we gave a basic methodology for recovering a system to a valid state when a run-time configuration compromises its architectural style. The main objective of our approach in [11] can be observed in Example 14 below.

Example 14 ([11]) *Consider the run-time reconfiguration*



where S changes as illustrated to model a failure F. By imposing an invariant that states that every client has to be connected to a non-failed server, the invalid configuration can be identified and recovered.

The fact that we can now record the evolution of our graphs gives us many advantages. The monitoring mechanism introduced in 3 and the way we apply reconfigurations now gives us the potential to identify which part of the graph (say G) has been re-written. Using the tree corresponding to the affected sub-graph (say G_R) one can identify the last production applied by observing the parent node of the leaves corresponding to G_R.

By using the monitoring system one can now spot where a violation may have occurred and attempt to fix such violations without considering or parsing the whole graph in every possible way.

Definition 10 (Parsing) *Given a sub-tree T_R of T we can identify the sub-graph G_R of G to be parsed. Let $p : L \xrightarrow{i} R$ be the production applied to obtain G_R as observed by its parent node in T_R we can compute the morphism $\sigma' : R \to G$ s.t. $\sigma'(R) = G_R$. Given an instance L' of L through the isomorphism $\iota : L \to L'$ such that $E_{L'} \cap E_G = \emptyset$ and $V_{L'} \cap V_G = \emptyset$ a graph $G' = G \setminus (E_{G_R} \cup \sigma'(R^\circ)) \cup L''$ (R° refer to the internal nodes of R) is the graph obtained by parsing G with p under the morphism σ' iff $L'' = L'[\iota(l) \mapsto \sigma'(i(l)) \mid l \in Im(i)]$.*

The methodology consists of the steps $1 \div 7$ below that require the designer to specify the productions and an architectural invariant ϕ_{inv} so to establish the architectural style of interest (as done in Example 14). Steps $2 \div 4$ are inherited from [11].

1. The architecture (say G) corresponding to the configuration of the current system is computed through ADR parsing as explained in [4]) and also formally defined in Def. 10.
2. Check that G satisfies ϕ_{inv} ($G \models \phi_{\text{inv}}$).
3. If $G \not\models \phi_{\text{inv}}$ then, for each production p, compute the weakest pre-condition ψ with respect to ϕ_{inv}.
4. Select a production $p : L \to R$ such that $G \backslash \sigma(E_L) \models \psi$ (if any); apply p to G to determine the reconfiguration needed for the system to reach a valid state.
5. If the designer considers not satisfactory the reconfigured system obtained in the previous stage or if there is no production p such that $G \backslash \sigma(E_L) \models \psi$, then the designer may repeat steps 3 and 4 by replacing G with $G \backslash \sigma(E_L)$ and ϕ_{inv} with ψ.
6. If no sequence of productions can recover the architectural style, we parse the graph at the point that has been violated as it is specified by the monitoring mechanism defined in § 3.
7. We now repeat steps $3 \div 7$ until the new graph $G' \models \psi$

In step 1, we rely on the *parsing* mechanism of ADR (cf. [4]) and Def. 10 whereby productions can be used "backward" to retrieve the architecture of a configuration. In step 2, we assume that an underlying monitoring mechanism uses the $\models$ relation of our logic to determine if the graph G computed in step 1 violates the invariant. In such case, step 3 uses the weakest pre-condition algorithm [11] on each production to compute their weakest pre-conditions (this step does not need to be re-iterated at each reconfiguration). In step 4, if the graph representing the violated system satisfies one of the computed weakest pre-conditions then the corresponding production is a candidate to re-establish the architectural style and trigger the appropriate reconfigurations on the invalid system. In step 5 the designer has to decide whether to stop or continue the process. In the latter case, the idea is to repeat steps 3 and 4 replacing G with $G \backslash \sigma(E_L)$ and ϕ_{inv} with ψ so to compute the weakest pre-condition of the weakest pre-condition computed in the previous iteration. This, allows us to exploit every possible sequence of productions that can be applied in order to enforce the architectural style. Note that the morphism that invalidates $G \models \phi_{\text{inv}}$ indicates which part of the system has to be rewritten, while the production p suggests plausible reconfigurations. If the above steps do not prove satisfactory then in step 6 the designer can use the tracking tree computed using Def. 7 and the function $\sqsubset$ in § 3 to identify the sub-graph that represents the reconfigured part of G. We then parse (see Def. 10) the affected sub-graph of G using the production that generated the sub-graph in the first place (production can be tracked from the tracking tree). Finally, in step 7 the designer has to decide whether to stop the process or repeat steps $3 \div 7$.

6 Conclusion

In this paper we defined a framework that allows us to exploit the "hierarchical nature" of ADR graphs. In particular, we used the functional reading of ADR productions as well as its reconfigurations that preserve the identity of components throughout the rewriting. Our framework permits to tackle certain architectural aspects of the design and allows the designer to identify and address problems at the architectural level.

Also, we refine the methodology proposed in [11] to automatically compute possible reconfigurations that recover from architectural style violations. Our refinement iteratively computes the weakest precondition to find a possible sequence of reconfigurations (if any) that re-establishes the architectural style of the system.

In this paper we focused on the development of the technical presentation of our framework. We proposed a monitoring mechanism through which the evolution of a computation is recorded and maintained in a tree-like structure reflecting the hierarchical nature of ADR graphs. We then exploit this mechanism to formally define more efficient parsing algorithms as well as more efficient ways of applying reconfigurations. Interestingly, this approach brings forth the definition of a new rewriting mechanism for ADR reconfiguration rules. More precisely, instead of parsing an ADR graph searching for a sub-graph matching the RHS of a reconfiguration rule, we propose a rewriting mechanism that visits the trees describing the graph evolution to find the match. We argue that this is more efficient than parsing the graph at the negligible cost of recording the evolution of the system through the monitoring. One could argue that the run-time monitoring of the system could be inefficient. In this respect, we note that a form of monitoring is necessary when dealing with self-configuring or self-healing systems; quoting [9] "autonomic system might continually monitor its own use". Since autonomic systems are the reference systems of this work, and, as a matter of fact, a form of monitoring is indispensable in order to identify run-time violation of the style, we argue that our approach simply adds to the necessary monitoring activities the cost of tracking the evolution of systems.

This paper completes and refines the framework initially proposed in [11]. We contemplate a new research direction to devise autonomic systems where component managers use architectural elements in the reconfigurations they distill. From this point of view, ADR is particularly suitable due to the fact that it can represent not only architectural level aspects of systems, but it can also be used to represent operational semantics by e.g. encodings of process calculi or modelling languages [3, 2]. This would immediately establish a connection between DbC approaches of abstractions levels close to the implementation of systems (for instance, see [1, 10]) with the DbC approach for ADR suggested in [11].

We conclude by commenting about the linearity conditions imposed on the reconfigurations rules (c.f. (6) and Def. 9). The linearity of the LHS of a reconfiguration rule can be relaxed at the cost of making the semantic of the matching more complex since multiple occurrences of the same variable would account for checking the existence of an isomorphism among different subgraphs. Instead, the linearity condition on the RHS of a production can be relaxed by simply using the counterpart semantic mechanism described in [5] to keep track of one copy of the variable. Finally, we note that in [4], reconfigurations rules of the form $r(x) : p(y) \to q(x,y)$ are considered, where x act as a parameter of the rule that can be used in its LHS or RHS. Such rules can be easily added to our framework using Def. 9 by mapping $\eta(x)$ to the fresh input graph.

References

[1] Laura Bocchi, Kohei Honda, Emilio Tuosto & Nobuko Yoshida (2010): *A Theory of Design-by-Contract for Distributed Multiparty Interactions*. In: *CONCUR*, pp. 162–176.

[2] Roberto Bruni, Howard Foster, Alberto Lluch-Lafuente, Ugo Montanari & Emilio Tuosto (2011): *A Formal Support to Business and Architectural Design for Service-Oriented Systems*. In Martin Wirsing & Matthias M. Hölzl, editors: *Results of the SENSORIA Project*, *Lecture Notes in Computer Science* 6582, Springer, pp. 133–152, doi:10.1007/978-3-642-20401-2_7.

[3] Roberto Bruni, Fabio Gadducci & Alberto Lluch Lafuente (2010): *An Algebra of Hierarchical Graphs and its Application to Structural Encoding*. *Scientific Annals of Computer Science* 20, pp. 53–96.

[4] Roberto Bruni, Alberto Lluch-Lafuente, Ugo Montanari & Emilio Tuosto (2008): *Style-Based Architectural Reconfigurations*. In: *Bulletin of the EATCS*, pp. 161–180.

[5] Fabio Gadducci, Alberto Lluch-Lafuente & Andrea Vandin (2012): *Counterpart Semantics for a Second-Order Calculus*. *Fundam. Inform.* 118(1-2), pp. 177–205, doi:10.3233/FI-2012-709.

[6] Dina Goldin, Scott A. Smolka & Peter Wegner, editors (2006): *Interactive Computation*. Springer-Verlag.

[7] Markus C. Huebscher & Julie A. McCann (2008): *A survey of autonomic computing - degrees, models, and applications*. *ACM Comput. Surv.* 40(3), pp. 7:1–7:28.

[8] Bart Jacob, Richard Lanyon-Hogg, Devaprasad K. Nadgir & Amr F. Yassin (2004): *A Practical Guide to the IIBM Autonomic Computing Toolkit*.

[9] Jeffrey O. Kephart & David M. Chess (2003): *The Vision of Autonomic Computing*. *Computer* 36(1), pp. 41–50, doi:10.1109/MC.2003.1160055.

[10] Yi Liu & H. Conrad Cunningham (2002): *Software Component Specification Using Design by Contract*. In: *Proceeding of the South East Software Engineering Conference*.

[11] Kyriakos Poyias & Emilio Tuosto (2012): *Enforcing Architectural Styles in Presence of Unexpected Distributed Reconfigurations*. In: *ICE*, pp. 67–82, doi:10.4204/EPTCS.104.7.

[12] Richard N. Taylor, Nenad Medvidović & Eric M. Dashofy (2009): *Software Architecture: Foundations, Theory, and Practice*. John Wiley and Sons.