A Graph Transformation Approach to Architectural Run-Time Reconfiguration

Michel Wermelinger
Departamento de Informática
Fac. de Ciências e Tecnologia
Universidade Nova de Lisboa
2825-114 Caparica, Portugal
mw@di.fct.unl.pt

Antónia Lopes and José Luiz Fiadeiro
Departamento de Informática
Faculdade de Ciências
Universidade de Lisboa
Campo Grande, 1700 Lisboa, Portugal
mal@di.fc.ul.pt jose@fiadeiro.org

Abstract

The ability of reconfiguring software architectures in order to adapt them to new requirements or a changing environment has been of growing interest. We propose a uniform algebraic approach that improves on previous formal work in the area due to the following characteristics. First, components are written in a high-level program design language with the usual notion of state. Second, the approach deals with typical problems such as guaranteeing that new components are introduced in the correct state (possibly transferred from the old components they replace) and that the resulting architecture conforms to certain structural constraints. Third, reconfigurations and computations are explicitly related by keeping them separate. This is because the approach provides a semantics to a given architecture through the algebraic construction of an equivalent program, whose computations can be mirrored at the architectural level.

1 Introduction

1.1 Motivation

One of the topics which is raising increased interest in the Software Architecture (SA) community is the ability to specify how a SA evolves over time, in particular at run-time, in order to adapt to new requirements or new environments, to failures, and to mobility. There are several issues at stake, among them:

modification time and source Architectures may change before execution, or at run-time (called dynamic reconfiguration). Run-time changes may be triggered by the current state or topology of the system (called programmed reconfiguration [6]) or may be requested unexpectedly by the user (called ad-hoc reconfiguration [6]).

modification operations The four fundamental operations are addition and removal of components and connections. Although their names vary, those operators are provided by most reconfiguration languages (like [6, 15, 1]). In programmed reconfiguration, the changes to perform are given with the initial architecture, but they may be executed when the architecture has already changed. Therefore it is necessary to query at run-time the state of the components and the topology of the architecture.

modification constraints Often changes must preserve several kinds of properties: structural (e.g., the architecture has a ring structure), functional, and behavioural (e.g., real-time constraints).

system state The new system must be in a consistent state.

1.2 Related Work

There is a growing body of work on architectural reconfiguration, some of it related to specific Architecture Description Languages (ADL), and some of formal, ADL-independent nature. Most of the proposals exhibit one of the following drawbacks.

- Arbitrary reconfigurations are not possible: Darwin [13] only allows component replication; ACME [18] only allows optional components and connections; Wright [1] requires the number of distinct configurations to be known in advance; [11] use context-free reconfiguration rules, which does not permit to create a new connection between exiting components, for example.

- The languages to represent computations are very simple and at a low level: rewriting of labels [11], process calculi [16, 2, 1], term rewriting [20, 8], graph rewriting [19]. They do not capture some of the abstractions used by programmers and often lead to cumbersome specifications.

- The combination of reconfiguration and computation, needed for run-time change, leads to additional formal constructs: [11] uses constraint solving, [16, 1, 2] define new semantics or language constructs for the process calculi, [8] must dynamically change the rewriting strategies, [19] imposes many constraints on the form of graph rewrite rules because they are used to express computation, communication, and reconfiguration. This often results in a proposal that is not very uniform, or has complex semantics, or does not make the relationship between reconfiguration and computation very clear.

1.3 Approach

To overcome these disadvantages, we have proposed an algebraic framework [22] using categorical diagrams to represent architectures, the double-pushout graph transformation approach[1] [5] to describe reconfigurations, and a program design language with explicit state to describe computations.

In this paper we refine our approach, introducing the notions of productive reconfiguration step and architectural style. To accommodate the latter, we have made the underlying mathematical definitions (not shown in this extended abstract) more uniform, based on the category of typed graphs [4], a generalisation of labelled graphs. Moreover, we cope with ad-hoc reconfiguration.

The running example is an airport luggage distribution system. One or more carts move continuously in the same direction on a N-units long circular track. A cart advances one unit at each step. Carts must not bump into each other. This is achieved by changing the movement interactions between carts, depending on their location. Reconfigurations may be due not only to mobility but also to component upgrade: a cart may be replaced by one with a built-in lap counter.

[1]To make the paper self-contained, the appendix contains an informal summary of the needed mathematical definitions.

2 CommUnity

2.1 Programs

CommUnity [7] is a parallel program design language based on Unity [3] and IP [9]. A program consists of a set of typed input and output variables, a boolean expression to be satisfied by the initial values of the output variables, and a set of actions, each of the form *name*: *guard* → *assignment(s)*. Action names act as *rendez-vous* points for program synchronisation (see Section 3). The empty set of assignments is denoted by `skip`. At each step, one of the actions is selected and, if its guard—a boolean expression over the variables—is true, its assignments are executed simultaneously. The values of the input variables are given by the environment and may change at each step. Input variables may not be assigned to by the program.

The next program describes the behaviour of a cart.

```
prog Cart
out    l : int
init   $0 \leq \mathrm{l} < N$
do     move: true $\rightarrow$ l := (l + 1) mod $N$
```

Henceforth we abbreviate "(l + 1) mod N" as "l $+_N$ 1" and omit the action guards when they are "true".

To take program state into account, we introduce a fixed set of typed variables, called logical variables. For the rest of the paper, it is the set $\{i, j : \text{int}, n : \text{nat}\}$. A program instance is then defined as a program together with a valuation function that assigns to each output variable a term (over logical variables) of the same type. No valuation is assigned to input variables because those are not under control of the program. Notice also that the valuation may return an arbitrary term, not just a ground term. Although in the running system the value of each variable is given by a ground term, we need variables to be able to write reconfiguration rules whose left-hand sides match components with possibly infinite distinct combinations of values for their variables. We represent program instances in tabular form (see below).

2.2 Superposition

A morphism from a program P to a program P' states that P is a component of the system P' and, as shown in [7], captures the notion of program superposition [3, 9]. Mathematically speaking, the morphism maps each variable of P into a variable of P' of the same type—such that output variables of the component P are mapped to output variables of the system

P'—and it maps each action name a of P into a (possible empty) set of action names $\{a'_1, \ldots, a'_n\}$ of P' [21]. Those actions correspond to the different possible behaviours of a within the system P'. Thus each action a'_i must preserve the functionality of a, possibly adding more things.

The next diagram shows in which way program "Cart" can be superposed with a counter that checks how often the cart passes by its start position. Notice how the second program strengthens the initialisation condition and it divides action "move" in two sub-cases.

$$\begin{tikzcd} \texttt{prog}\ \text{Cart}\ \ldots \arrow[d, "move\mapsto\{move,pass\}"', "l\mapsto loc"] \\ \texttt{prog}\ \text{CartWithLaps} \end{tikzcd}$$

```
prog CartWithLaps
out    loc, sloc, laps : int
init   0 ≤ loc < N ∧ sloc = loc ∧ laps = 0
do     pass: loc +_N 1 = sloc
       → loc := loc +_N 1 || laps := laps + 1
[]     move: loc +_N 1 ≠ sloc → loc := loc +_N 1
```

A morphism between program instances is simply a superposition morphism that preserves the state. To be more precise, if an output variable of P is mapped to an output variable of P', their valuations must be the same for any substitution of the logical variables. An example is

$$\begin{tikzcd} \begin{array}{|c|c|}\hline \multicolumn{2}{|c|}{\text{Cart}} \\ \hline \text{l} & i \\ \hline \end{array} \arrow[r, "l\mapsto loc", "move\mapsto\{move,pass\}"'] & \begin{array}{|l|l|}\hline \multicolumn{2}{|c|}{\text{CartWithLaps}} \\ \hline \text{l} & 1 * i \\ \text{sloc} & i +_N 1 \\ \text{laps} & n + 1 \\ \hline \end{array} \end{tikzcd}$$

where the instance on the right represents a cart that has completed at least one lap and will complete another one in the next step.

3 Architectures

3.1 Configurations

Interactions between programs are established through action synchronisation and memory sharing. This is achieved by relating the relevant action and variable names of the interacting programs.

The categorical framework imposes the locality of names. To state that variable (or action) a_1 of program P_1 is the same as variable (resp. action) a_2 of P_2 one needs a third, "mediating" program C—the channel—containing just a variable (resp. action) a and two morphisms $\sigma_i : C \to P_i$ that map a to a_i. A channel has no computations of its own. Therefore it has no output variables (hence no assignments nor initialisation condition) and all actions have true guards. We abbreviate a channel as $\langle I \mid A \rangle$, where I is the set of input variables and A is the set of action names.

Problems arise if two synchronised actions update a shared variable in distinct ways. As actions only change the values of output variables, it is sufficient to impose that output variables are not shared, neither directly through a single channel nor indirectly through a sequence of channels. We call such diagrams configurations. This restriction forces interactions between programs to be synchronous communication of values (from output to input variables), a very general mode of interaction that is suitable for the modular development of reusable components, as needed for architectural design.

It can be proved that every finite configuration has a colimit, which returns the minimal program that simulates the execution of the overall system. Briefly put, the colimit is obtained by taking the disjoint union of the variables (modulo shared variables), the cartesian product of actions (modulo synchronized ones)—to denote parallel execution of non-synchronised actions—, and the conjunction of the initialisation conditions. Actions are synchronized by taking the conjunction of the guards and the parallel composition of assignments. An example is provided in the next section.A configuration instance is a configuration whose nodes are program instances. Since output variables are not shared, they have no conflicting valuations. Therefore every configuration instance has a colimit, given by the colimit of the underlying configuration together with the union of the valuations of the program instances.

3.2 Connectors

SA has put forward the notion of connector to encapsulate the interactions between components. An n-ary connector consists of n roles R_i and one glue G stating the interaction between the roles. These act as "formal parameters", restricting which components may be linked together through the connector. We represent a connector by a diagram of the form

$$\begin{tikzcd} & C_1 \arrow[dl, "\gamma_1"'] \arrow[r, "\rho_1"] & R_1 \\ G & \vdots & \vdots \\ & C_n \arrow[ul, "\gamma_n"] \arrow[r, "\rho_n"] & R_n \end{tikzcd}$$

where the channels indicate which variables and actions of the roles are used in the interaction specification, i.e., the glue. An n-ary connector can be applied to components $P_1, \ldots, P_n$ when morphisms $\iota_i : R_i \to P_i$ exist. This corresponds to the intuition that the "actual arguments" (i.e., the components) must instantiate the "formal parameters" (i.e., the roles).

An architecture (instance) is then a configuration (instance) where all components interact through connectors, and all roles are instantiated. Hence any architecture has a semantics given by its colimit, which returns the minimal program that simulates the execution of the overall system.

To avoid a cart c_1 colliding with the cart c_2 right in front of it we only need to make sure that if c_1 moves, so must c_2, but the opposite is not necessary. We say action a subsumes action b if b executes whenever a does. This can be seen as a partial synchronisation mechanism: a is synchronised with b, but b can still execute freely. The diagram in Figure 1 shows the application of the generic action subsumption connector to two carts and the resulting colimit. Notice that although the two roles are isomorphic, the binary connector is not symmetric because the channel morphisms and the glue treat the two actions differently: "b" may be executed alone at any time, while "a" must co-occur with "b".

3.3 Style

In general, a role may be instantiated by different components, and it may be even the case that the same component can instantiate the same role in different ways (e.g., if 'Cart' had other actions). But normally only a few of all the possibilities are meaningful to the application at hand. The allowed ways to apply connectors to components can be described by typed graphs. This leads to a declarative notion of architecture style: it consists of a set of components, a set of connectors, and a diagram T in the category of programs and superposition morphisms using only those connectors and components. Every architecture written by the user must then come equipped with a morphism to T proving that it obeys the restrictions imposed by T. As for an architecture instance, it is well-typed if the underlying architecture, obtained by forgetting the valuations, is. We believe that this approach to architectural styles, besides being simple to use, is also sufficient in many occasions, namely when only the kinds of interactions between the given components have to be restrained. Abstract architectural patterns (e.g., pipe-filter, layer) cannot be described with our approach.

For our example, the set of components is 'Cart' and 'CartWithLaps', the set of connectors is just the action subsumption connector shown before, and the architecture type T (with morphisms as shown in previous diagrams) is

$$\begin{tikzcd}
\langle \mid \mathrm{a}\rangle \arrow[r] \arrow[d] & \mathrm{Subsume} & \langle \mid \mathrm{b}\rangle \arrow[l] \arrow[d] \\
\mathrm{Subsumer} \arrow[r] & \mathrm{Cart} \arrow[d] & \mathrm{Subsumed} \arrow[l] \\
 & \mathrm{CartWithLaps} &
\end{tikzcd}$$

stating that the connector may be applied to carts only, which in turn may be refined with a lap counter.

Notice that a style T, by showing all possible morphisms that may occur in an architecture, also restricts the visibility of variables, stating which output variables are to be shared (and how) and which are private to each program.

It is important to notice that T is not necessarily a configuration: since it shows in a single diagram all morphisms that may occur in architectures, it may happen that output variables are shared in T.

4 Dynamic Reconfiguration

Basically, we represent dynamic reconfiguration as a rewriting process over graphs with nodes labelled by program instances and arcs labelled by instance morphisms. In essence, a reconfiguration rule is a graph production, and a reconfiguration step is a direct derivation. This ensures that the state of components and connectors that are not affected by a rule does not change, because node labels (which include the variables' valuations) are preserved, thus keeping reconfiguration and computation separate. However, we must make slight adaptations of the basic graph transformation framework to our setting.

First, in the double-pushout approach, there is no restriction on the obtained graphs, but in reconfiguration we must check that the result is indeed an architecture, otherwise the rule (with the given match) is not applicable. Without this restriction, it would be possible for a rule to introduce a connector that would lead to sharing of output variables, for example.

Second, it should not be possible to apply the same rule in the same way (i.e., to the same program instances) more than once because that would lead to infinite reconfiguration sequences. To this end we restrict the allowed reconfiguration sequences by considering only productive direct derivations $G \overset{p,m}{\Longrightarrow} H$: there are no graph morphisms $lr : L \to R$ and $x : R \to G$ such that $lr; x = m$. The existence of lr shows that production p does not delete any nodes or arcs. The remaining conditions check that the match is being applied to a part of G that corresponds to the right-hand side R

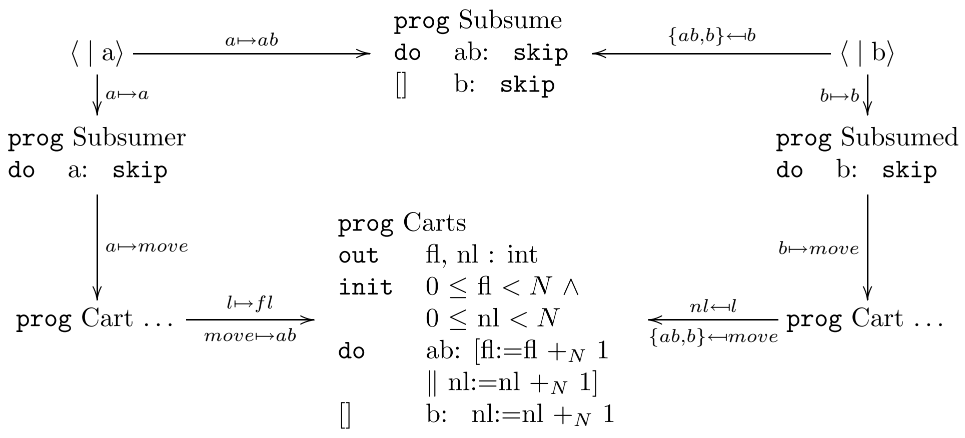


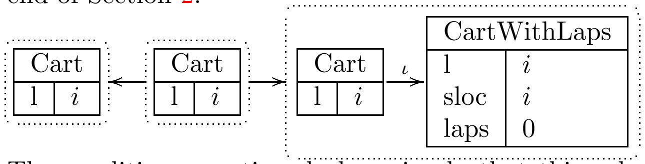
Figure 1. An applied action subsumption connector and its colimit

and therefore can have been generated by a previous application of this production. Our definition is a particular case of productions with application conditions [10]: a derivation is productive if p is applicable to G using the negative application condition lr.

Third, dynamic reconfiguration rules must be conditional, because they depend on the current state. Thus they are of the form $L \xleftarrow{l} K \xrightarrow{r} R$ **if** B, with B a proposition over the logical variables occurring in L. Moreover, a rule can only be applied if every new component added by the rule is in a precisely determined state that satisfies the initialisation condition, in order to be able to perform computations right away. For that purpose, we require that the logical variables occurring in R also occur in L. The definition of reconfiguration step must be changed accordingly. At any point in time the current system is given by an architecture instance whose valuations return ground terms. Therefore the notion of matching must also involve a compatible substitution of the logical variables occurring in the rule by ground terms. If we apply the substitution to the whole rule, we obtain a rule without logical variables that can be directly applied to the current architecture using the normal definition of derivation as a double pushout over labelled graphs. However, the notion of state introduces two constraints. First, the substitution must obviously satisfy the application condition B. Second, the derivation must make sure that the state of each program instance added by the right-hand side satisfies the respective initialisation condition.

Returning to our example, to avoid collisions we give in Figure 2 a rule that applies the action subsumption connector to two carts that are less than 3 units apart, where the graph morphisms l and r are obvious. The opposite rule (with the negated condition) is necessary to remove the connector when no longer needed.

As a second example, if we want to add a counter to a cart, no matter which connectors it is currently linked to, we just unconditionally superpose the 'CartWithLaps' program on it, with ι the morphism shown at the end of Section 2:



The conditions mentioned above imply that this rule can only be applied with a substitution that satisfies $0 \le i \le N$. This example illustrates how to describe the transfer of state from old to new components. In this case it is just a copy of value i, but in general the right-hand side may contain arbitrarily complex terms that calculate the new values from the old ones.

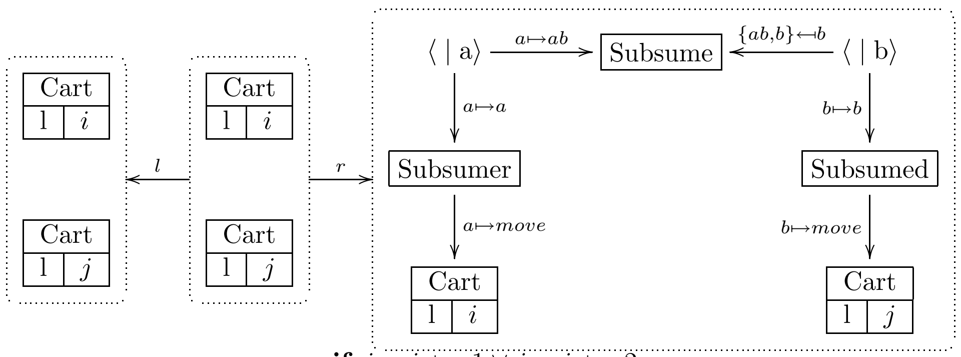
If there is an architectural style T, then the three architecture instances in a reconfiguration rule must be typed by T. It can be proved that the graph obtained through direct derivation is also well-typed.

To coordinate computations and reconfigurations, the run-time infrastructure executes the following sequence:

1. allow the user to change the style and the set of reconfiguration rules;

2. find a maximal sequence of reconfiguration steps starting with the current architecture instance A, obtaining A';

3. compute the colimit S of A';



if $j = i +_N 1 \vee j = i +_N 2$

Figure 2. Introduction of the action subsumption connector

4. if none of the S's actions can be executed, stop, otherwise update the values of S's variables according to the chosen action;

5. propagate through the colimit morphisms the changes back to the variables of the program instances of A', call the new diagram A, and go to step 1.

The first step caters for ad-hoc reconfiguration. In our example, it allows to add the CartWithLaps program to the style and to add the last rule shown. Step 5 keeps the state of the program instances in the architectural diagram consistent with the state of the colimit, and ensures that at each point in time the correct conditional rules are applied. As [14, 11] we adopt a two-phase approach: computations (step 4) are interleaved with reconfiguration sequences (step 2). In this way, the specification of the components is simpler, because it is guaranteed that the necessary interconnections are in place as soon as required by the state of the components.

5 Concluding Remarks

We have refined our algebraic foundation for dynamic software architecture reconfiguration. Our approach has several advantages over previous work [11, 16, 1, 2, 8, 20]:

- context-dependent rewriting allows arbitrary reconfigurations;

- computations (on a program) and reconfigurations (on an architecture) are explicitly related through a colimit operation, because we do not rewrite just graphs, but diagrams in a category of programs with superposition;

- the maintenance of state consistency during reconfiguration—how to transfer state, in which state reconfigurations are possible, what is the state of new components—is straightforward to specify, due to the use of a program design language that is more natural than terms, process calculi, or graphs, leading to easy to read rules.

The algebraic graph transformation approach combines well with our categorical framework for architectural design and has several advantages: it enforces that component state is only changed by computations, not by reconfiguration steps; the application conditions of the double-pushout approach enforce that components are not removed while linked to connectors, thus not leaving "dangling" roles (not shown in this abstract); the negative application conditions can be used to avoid useless changes to the architecture; typed graphs provide, besides a uniform mathematical basis, a declarative and simple notion of style—sufficient to describe certain structural modification constraints—that can be automatically maintained during reconfiguration.

References

[1] R. Allen, R. Douence, and D. Garlan. Specifying and analyzing dynamic software architectures. In *Fundamental Approaches to Software Engineering*, volume 1382 of *LNCS*, pages 21–37. Springer-Verlag, 1998.

[2] C. Canal, E. Pimentel, and J. M. Troya. Specification and refinement of dynamic software architectures. In *Software Architecture*, pages 107–125. Kluwer Academic Publishers, 1999.

[3] K. M. Chandy and J. Misra. *Parallel Program Design—A Foundation.* Addison-Wesley, 1988.

[4] A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamentae Informatica*, 26(3–4):241–266, 1996.

[5] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation, part I: Basic concepts and double pushout approach. Technical Report TR-96-17, University of Pisa, Mar. 1996.

[6] M. Endler. A language for implementing generic dynamic reconfigurations of distributed programs. In *Proceedings of the 12th Brazilian Symposium on Computer Networks*, pages 175–187, 1994.

[7] J. L. Fiadeiro and T. Maibaum. Categorial semantics of parallel program design. *Science of Computer Programming*, 28:111–138, 1997.

[8] J. L. Fiadeiro, M. Wermelinger, and J. Meseguer. Semantics of transient connectors in rewriting logic. Position Paper for the First IFIP Working International Conference on Software Architecture, Feb. 1999.

[9] N. Francez and I. Forman. *Interacting Processes.* Addison-Wesley, 1996.

[10] A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3–4), 1996.

[11] D. Hirsch, P. Inverardi, and U. Montanari. Modelling software architectures and styles with graph grammars and constraint solving. In *Software Architecture*, pages 127–143. Kluwer Academic Publishers, 1999.

[12] M. Löwe. Algebraic approach to graph transformation based on single pushout derivations. Technical Report 90/5, Technische Universität Berlin, Fachbereich 13, Informatik, 1990.

[13] J. Magee and J. Kramer. Dynamic structure in software architectures. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 3–14. ACM Press, 1996.

[14] P. J. McCann and G.-C. Roman. Compositional programming abstractions for mobile computing. *IEEE Transactions on Software Engineering*, 24(2), Feb. 1998.

[15] N. Medvidovic. ADLs and dynamic architecture changes. In *Joint Proceedings of the SIGSOFT'96 Workshops*, pages 24–27. ACM Press, 1996.

[16] D. L. Métayer. Describing software architecture styles using graph grammars. *IEEE Transactions on Software Engineering*, 24(7):521–553, July 1998.

[17] J. C. Mitchell. *Foundations for Programming Languages.* MIT Press, 1996.

[18] R. T. Monroe, D. Garlan, and D. Wile. *Acme Straw-Manual*, Nov. 1997.

[19] G. Taentzer, M. Goedicke, and T. Meyer. Dynamic change management by distributed graph transformation: Towards configurable distributed systems. In *Proc. 6th Int. Workshop on Theory and Application of Graph Transformation*, 1998.

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

[21] M. Wermelinger and J. L. Fiadeiro. Connectors for mobile programs. *IEEE Transactions on Software Engineering*, 24(5):331–341, May 1998.

[22] M. Wermelinger and J. L. Fiadeiro. Algebraic software architecture reconfiguration. In *Software Engineering—ESEC/FSE'99*, volume 1687 of *LNCS*, pages 393–409. Springer-Verlag, 1999.

A Mathematical Definitions

A.1 Category Theory

Category Theory [17] is the mathematical discipline that studies, in a general and abstract way, relationships between arbitrary entities. A category is a collection of objects together with a collection of morphisms between pairs of objects. A morphism f with source object a and target object b is written $f : a \to b$ or $a \xrightarrow{f} b$. Morphisms come equipped with a composition operator ";" such that if $f : a \to b$ and $g : b \to c$ then $f; g : a \to c$. Composition is associative and has identities id_a for every object a.

Diagrams are directed graphs—where nodes denote objects and arcs represent morphisms—and can be used to represent "complex" objects as configurations of smaller ones. For categories that are well behaved, each configuration denotes an object that can be retrieved through an operation on the diagram called colimit. Informally, the colimit of a diagram returns the "minimal" object such that there is a morphism from every object in the diagram to it (i.e., the colimit contains the objects in the diagram as components) and the addition of these morphisms to the original configuration results in a commutative diagram (i.e., interconnections, as established by the morphisms of the configuration diagram, are enforced).

Pushouts are colimits of diagrams of the form $b \xleftarrow{f} a \xrightarrow{g} c$. By definition of colimit, the pushout returns an object d such that the diagram

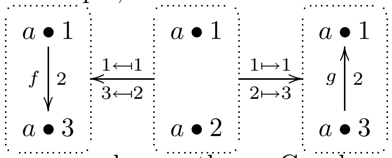
$$\begin{tikzcd} & a \arrow[dl, "f"'] \arrow[dr, "g"] \arrow[dd, "i"] & \\ b \arrow[dr, "h"'] & & c \arrow[dl, "j"] \\ & d & \end{tikzcd}$$

exists and commutes (i.e., $f; h = i = g; j$). Furthermore, for any other pushout candidate d', there is a unique morphism $k : d \to d'$. This ensures that d, being a component of any other object in the same conditions, is minimal. Object c is called the pushout complement of diagram $a \xrightarrow{f} b \xrightarrow{h} d$.

A.2 Graph Transformation

The algebraic approach to graph transformation [5] was introduced over 20 years ago in order to generalize grammars from strings to graphs. Hence it was necessary to adapt string concatenation to graphs. The approach is algebraic because the gluing of graphs is done by a pushout in an appropriate category. There are two main variants, the double-pushout approach [5] and the single-pushout approach [12]. We only use the former. It is based on a category whose objects are labelled graphs and whose morphisms $f : a \to b$ are total maps (from a's nodes and arcs to those of b) that preserve the labels and the structure of a.

A graph transformation rule, called graph production, is simply a diagram of the form $L \xleftarrow{l} K \xrightarrow{r} R$ where L is the left-hand side graph, R the right-hand side graph, K the interface graph and l and r are injective graph morphisms. The rule states how graph L is transformed into R, where K is the common subgraph, i.e., those nodes and arcs that are not deleted by the rule. As an example, the rule



substitutes an arc by another. Graphs are written within dotted boxes to improve readability. Nodes and arcs are numbered uniquely within each graph to show the mapping done by the morphisms.

A production p can be applied to a graph G if the left-hand side can be matched to G, i.e., if there is a graph morphism $m : L \to G$. A direct derivation from G to H using p and m exists if the diagram

$$\begin{tikzcd} L \arrow[d, "m"] & K \arrow[l, "l"'] \arrow[r, "r"'] \arrow[d, "d"] & R \arrow[d, "m^*"] \\ G & D \arrow[l, "l^*"] \arrow[r, "r^*"] & H \end{tikzcd}$$

can be constructed, where each square is a pushout. Intuitively, first the pushout complement D is obtained by deleting from G all nodes and arcs that appear in L but not in K. Then H is obtained by adding to D all nodes and arcs that appear in R but not in K. The fact that l and r are injective guarantees that H is unique. An example derivation using the previously given production is Figure 3.

A direct derivation is only possible if the match m obeys two conditions. First, if the production removes a node $n \in L$, then each arc incident to $m(n) \in G$ must be image of some arc attached to n. Second, if the production removes one node (or arc) and maintains another one, then m may not map them to the same node (or arc) in G.

Two examples in which the match violates these conditions are represented by the following diagrams, where $\emptyset$ is the empty graph.

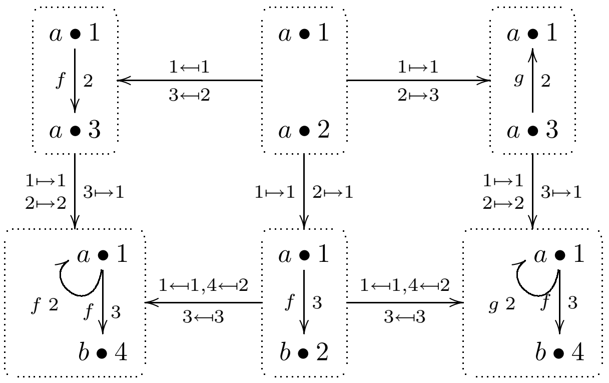
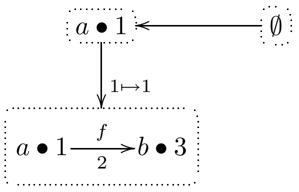
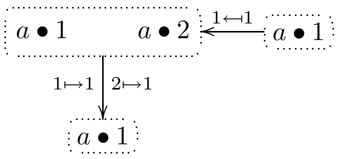


Figure 3. Applying a graph production





Both conditions are quite intuitive. The first one prevents dangling arcs, the second one avoids contradictory situations. Both allow an unambiguous prediction of removals. A node of G will be removed only if its context (i.e., adjacent arcs and nodes) are *completely* matched by the left-hand side of some production. The advantage is that the production specifier can control exactly in which contexts a node is to be deleted. This means it is not possible to remove a node no matter what other nodes are linked to it.