A Graph based architectural (re)configuration language

A Graph Based Architectural (Re)configuration Language[*]

Michel Wermelinger
Departamento de Informática
Fac. de Ciências e Tecnologia
Universidade Nova de Lisboa
2829-516 Caparica
Portugal
http://ctp.di.fct.unl.pt/˜mw

Antónia Lopes
Dep. de Informática
Faculdade de Ciências
Universidade de Lisboa
Campo Grande
1700 Lisboa, Portugal
http://www.di.fc.ul.pt/˜mal

José Luiz Fiadeiro
Dep. de Informática
Faculdade de Ciências
Universidade de Lisboa
Campo Grande
1700 Lisboa, Portugal
http://www.fiadeiro.org/jose

ABSTRACT

For several different reasons, such as changes in the business or technological environment, the configuration of a system may need to evolve during execution. Support for such evolution can be conceived in terms of a language for specifying the dynamic reconfiguration of systems. In this paper, continuing our work on the development of a formal platform for architectural design, we present a high-level language to describe architectures and for operating changes over a configuration (i.e., an architecture instance), such as adding, removing or substituting components or interconnections. The language follows an imperative style and builds on a semantic domain established in previous work. Therein, we model architectures through categorical diagrams and dynamic reconfiguration through algebraic graph rewriting.

1. INTRODUCTION

One of the topics that is raising increased interest in the Software Architecture community is the ability to specify how an architecture evolves over time, in particular at run-time, in order to adapt to new requirements, new business rules or new environments, to failures, and to mobility. This topic raises several issues [26], among which:

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 [7]) or may be requested unexpectedly by the user (called ad-hoc reconfiguration [7]).

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 [7, 20, 1, 15]). 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.

There is a growing body of work on architectural reconfiguration, some of it related to specific Architecture Description Languages (ADL) [4, 18, 25, 1], and some of a formal, ADL-independent nature [13, 19, 22, 28]. As we argued in [29], most of the proposals exhibit one of the following drawbacks.

- Arbitrary reconfigurations are not possible (e.g., only component replication is allowed).
- The languages used for the representation of computations are at a low level of abstraction (e.g., process calculi). 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. This often results in a proposal that is not uniform, or has complex semantics, or does not make the relationship between reconfiguration and computation clear enough.

To overcome these disadvantages, we have proposed in the last ESEC/FSE an algebraic approach [29], using categorical diagrams to represent architectures, the double-pushout graph transformation approach [6] to specify reconfigurations, and a program design language with explicit state to describe computations [17]. In [30] we have further refined and extended the work, using typed graphs to represent simple topological invariants for the reconfiguration process. We have shown that our approach has several advantages over previous work, avoiding the drawbacks mentioned above:

- Architectures, reconfigurations, and connectors are represented and manipulated in a graphical yet mathematical rigorous way at the same language-independent level of abstraction, resulting in a uniform framework based simply on diagrams and their colimits (a universal categorical construction).

[*]This research was partially supported by Fundação para a Ciência e Tecnologia through project POSI/32717/00 (FAST—Formal Approach to Software Architecture) and by ATX Software SA.

- The chosen program design language is at a high level of abstraction, allowing a more intuitive representation of program state and computations.

- Computations and reconfigurations are kept separate but related in an explicit, simple, and direct way through the colimit construction.

- Typed graphs provide a simple and declarative notion of modification constraints.

- Several practical problems—maintaining the constraints during reconfiguration, transferring the state during replacement, removing components in a quiescent state [15], adding components properly initialized—are easily handled.

However, the work presented was meant as an investigation into solid formal foundations for dynamic reconfiguration. It was not meant as an actual specification language. In fact, as argued in [27], describing certain kinds of transformations using only graph rewriting rules can be quite cumbersome, because it leads to heavy use of dummy nodes and arcs to control the exact way in which rewriting rules are to be applied. Unfortunately, ADLs either do not handle reconfiguration (like [24]) or they provide only a few basic reconfiguration constructs (like [25, 1, 4, 19, 20]) making it hard or impossible to express moderately complex reconfigurations. Therefore we turned to work in Distributed Systems, in which the technological aspects of reconfiguration have been investigated for a long time.

Two of the most elegant and expressive approaches we found are Gerel [7] and reconfiguration programming [23]. Both allow the description of very complex reconfigurations in a relatively easy way, combining the usual programming language constructs (like selection and iteration) with specialised reconfiguration commands. However, approaches like these lack a formal basis, and they do not have an explicit notion of the system's software architecture in which the reconfiguration occurs. Taking a distributed systems perspective, some approaches describe reconfigurations at a low level of abstraction, usually (dis)connecting components on a port by port basis.

In this paper we try to make a bridge between the rigour of formal approaches, the abstractions of Software Architecture approaches, and the pragmatics of Distributed Systems approaches, by presenting a language to describe architectures and complex reconfigurations in a straightforward way, while keeping the algebraic foundations laid out in our previous work. We are more interested in what are the necessary constructions needed for an effective but formal reconfiguration language, than in the actual language itself. Therefore, this paper only presents a minimal language with one specialised construct for each task (iteration, removing components, interconnecting components, querying the configuration, etc.).

The running example is taken from banking. We consider customers and accounts, in a one-to-one relationship. Besides a normal kind of accounts, there may also exist salary accounts, in which its owner gets her/his salary deposited every month. For normal accounts, it is not possible to withdraw more than the current balance, but a salary account may be overdrawn up to the salary amount. Of course, such a benefit has a price, to be paid in terms of interest calculated over the period and amount the account is below zero. Therefore, when such an account is created, or when an existing normal account becomes a salary account, the customer must explicitly state how much credit s/he wants. The credit must not exceed the salary, and in particular it may be zero, i.e., the customer may wish not to take advantage of having credit. The bank is not interested in accounts that have an *average* balance below zero. In those cases, the policy is to return those accounts automatically to their normal status, to prevent the financial situation of the account, and the debt of the customer towards the bank, of getting worse.

We start by summarising COMMUNITY, the program design language used to describe computations. We then introduce the language fragment to describe architectures and their constraints. Finally, in Section 4 we present the remaining of the language, in order to describe reconfigurations.

2. COMMUNITY

COMMUNITY programs are in the style of UNITY programs [5], but they also combine elements from IP [11]. However, COMMUNITY has a richer coordination model and, even more important, it requires interaction between components to be made explicit. In this way, the coordination aspects of a system can be separated from the computational aspects and externalised, making explicit the gross modularisation of the system in terms of its components and its interactions.

In this section, we summarise the syntax of COMMUNITY programs and introduce a notation for describing system configurations. Each configuration can be transformed into a single, semantically equivalent, program that represents the whole system. Finally, we discuss refinement of COMMUNITY programs. The formal definitions and proofs were presented in [17, 16].

2.1 Designs

COMMUNITY is independent of the actual data types used and, hence, we assume there are pre-defined sorts and functions given by a fixed algebraic signature in the usual sense. For the purpose of examples, we consider an algebraic signature containing sorts `bool` (booleans), `nat` (natural numbers) and `int` (integers) with the usual operations.

A COMMUNITY design consists of a set of typed variables and a set of actions. There are *input*, *output* and *private* variables. Input variables are read-only. Output and private variables are called *local* variables and cannot be changed by the environment. A design with input variables is *open* in the sense that it needs to be connected to other components of the system to read data, as explained in Section 2.2.

There are *private* and *shared* actions, but our example only uses the latter. Each action has a name, two guards, and a set of non-deterministic assignments. The *safety* and *progress* guards are propositions over the local variables and establish an interval in which the enabling condition of the action must lie, the safety guard being the lower bound. When the guards are equivalent we write only one of them. At each execution step, one of the actions whose enabling condition holds of the current state is selected, and its assignments are executed atomically in parallel. A non-deterministic assignment $l :\in E$ assigns to l one of the elements of set E. We abbreviate $v :\in \{t\}$ as $v := t$, and we write `skip` to denote the absence of assignments.

For our example, we need the following designs. The first simply describes the basic functionality of an account: it lets money to be deposited and withdrawn without any limits. There is also a variable to store the account number. No action changes the variable because its value remains fixed while the account exists. There is an action to calculate the average balance of the account, but for the moment it is irrelevant how and when the average is computed.

```
design NormalAccount
in     amount : nat
out    balance, avgbal, number : int
do     deposit: true → balance := balance + amount
[]     withdraw: true → balance := balance - amount
[]     average: true, false → avgbal :∈ int
```

The second design describes a salary account, which adds to 'NormalAccount' a variable to store the credit given to this account, i.e., the salary deposited in it every month. The safety guard of action 'withdraw' is changed accordingly.

```
design SalaryAccount
in     amount : nat
out    balance, avgbal, number, salary : int
do     deposit: true → balance := balance + amount
[]     withdraw: balance - amount ≥ -salary
       → balance := balance - amount
[]     average: true, false → avgbal :∈ int
```

The next design models the behaviour of a customer. S/he chooses some value and then invokes one of the two operations.

```
design Customer
out    value : nat
prv    ready : bool
do     deposit: ready, false → ready := false
[]     withdraw: ready, false → ready := false
[]     choose: ¬ready, false → value :∈ nat || ready := true
```

2.2 Configurations

In COMMUNITY, the model of interaction between components is based on action synchronisation and the interconnection of input variables of a component with output variables of other components. Although these are common forms of interaction, COMMUNITY, unlike other program design languages, requires interaction between components (name bindings) to be made explicit.

Connecting a design D_1 with a design D_2 is done through a *channel*: a set of bindings $i_{1,j}$-$i_{2,j}$, where each $i_{1,j}$ is a non-private variable or a set of shared actions of design D_1. In the first case, $i_{2,j}$ must be a non-private variable of D_2, of the same sort as $i_{1,j}$, and the pair $i_{1,1}$-$i_{2,1}$ denotes that the two variables are to be shared. In the second case, $i_{2,j}$ must be a set of shared actions of D_2. Moreover, every shared action of the involved designs can appear at most once in the channel definition. Intuitively, a pair

$$\{a_{1,1}, \ldots, a_{1,n}\}\text{-}\{a_{2,1}, \ldots, a_{2,m}\}$$

states that any action $a_{1,i}$ of D_1 must occur simultaneously (i.e., synchronise) with some action $a_{2,j}$ of D_2 and vice versa.

For example, the channel

{value-amount, deposit-deposit, withdraw-withdraw}

to connect a customer directly to a normal account, allows the customer to withdraw unlimited funds anytime, a highly undesirable situation (for the bank, of course).

A *configuration* is given by an undirected labelled graph, where each node is labelled by a design, and each arc by a channel, such that: 1) no node is connected to itself; 2) no two output variables are (directly or indirectly) shared.

Such a configuration has a very precise mathematical semantics, given by a diagram in a category whose objects are designs and whose morphisms capture a notion of program superposition [5]. An interconnection between two components is described by a design that just declares the common variables and actions of the components, without introducing any additional behaviour, thus corresponding to the recently proposed notion of duct [21]. This semantics has been presented previously [17, 16] and it is trivial to translate configurations into categorical diagrams. One of the advantages of this categorical semantics is that any diagram corresponding to a configuration can be transformed, by a universal categorical construction called *colimit*, into a single design that represents the whole system, as proven in [16].

A *run-time* configuration is a configuration in which each node, besides being labelled with a design D, is also labelled with its current state, i.e., with one pair $\langle l, value\rangle$ for each local variable l of D. Because local variables cannot be shared among designs, it is not possible for two different nodes to have different values for the same variable. Hence, the colimit of a run-time configuration always exists and is given by the colimit of the underlying configuration together with the disjoint union of all variable-value pairs.

2.3 Refinement

A key factor for architectural description is a notion of refinement that can be used to support abstraction. A design R refines design P if each variable of P is mapped to a variable of R of the same sort and kind (input, output or private), and each action of P is mapped to a set of actions of R of the same kind (private or shared), such that the functionality and interface (i.e., the "binding points") of P are preserved. The interface is preserved by requiring the mapping of input and output variables to be injective and the image of a shared action to be a non-empty set. To preserve functionality safety guards may not be weakened, progress guards may not be strengthened, and assignments may not be less deterministic.

The concrete syntax to define refinement morphisms is

```
refinement Name: $D_1 \to D_2$
$x_{1,1}$/$x_{2,1}$ ...
end refinement
```

such that each $x_{1,i}$ is a variable of design D_1—and in that case $x_{2,i}$ is a variable of D_2—or an action of D_1—and in that case $x_{2,i}$ is a set of actions of D_2. The identity and inclusion refinement morphisms are implicitly defined (see Section 3.4). For our example, we have the inclusion of 'NormalAccount' in 'SalaryAccount'. Another way of defining refinements is through composition:

```
refinement Name: $D_1 \to D_{n+1}$
Name$_1$ ; Name$_2$ ; ...; Name$_n$
end refinement
```

with *Name*$_i$: $D_i \to D_{i+1}$ a previously defined refinement or the reserved word `inclusion`.

Channels can be composed with refinement morphisms. To be more precise, given a channel between designs D_1 and D_2 and a refinement D_3 of D_2, one can calculate a channel between D_1 and D_3 by composing every binding $i_{1,j}$-$i_{2,j}$ with the mapping $i_{2,j}/i_{3,j}$. Moreover, the colimit of the obtained configuration is a refinement of the colimit of the original configuration.

3. SOFTWARE ARCHITECTURE

An architecture is a class of configurations that exhibit some conceptual unity. An architectural description must therefore include the designs and interactions to be used in those configurations and a set of restrictions on the possible configurations.

We start by describing a construct to specify complex interactions between components, using the primitive mechanisms of action synchronization and variable sharing given by channels. Next we show how to describe conditions on the run-time configurations and finally we present the concrete syntax for architectures.

3.1 Connectors

Although components have always been considered the fundamental building blocks of software systems, the way the components of a system interact may be also determinant on the system properties. Component interactions were recognised also to be first-class design entities and architectural connectors have emerged as a powerful tool for supporting the description of these interactions.

According to [2], an n-ary connector consists of n *roles* and one *glue* stating the interaction between the roles. The use of a connector in the construction of a particular system is realised by the instantiation of its roles with specific components of the system. Therefore the roles act as "formal parameters", restricting which components may be linked together through the connector.

In our framework, a connector is represented by a star-shaped configuration, with the glue, in the center, linked to the roles, in the points of the star [9]. Normally, different connectors may use the same designs for roles, because there may exist different kinds of interactions between the same kinds of components. Moreover, the roles are the "interface" of a connector, i.e., they must be publicly known in order to intantiate them with the actual components. For these two reasons, we require the designs used for roles to be specified before and outside the specification of connectors. A connector then just specifies the glue and the channels.

For our running example we need a connector with two roles, one to be instantiated with a customer and the other to be instantiated with any account. To make the example more compact, we use for the first role the 'Customer' design itself. The second role is

```
design AnyAccount
in     value : nat
out    balance : int
do     credit: true, false → balance :∈ int
[]     debit: true, false → balance :∈ int
```

As for the connector, its glue contains a private variable to store the amount of credit given to the customer. If the customer is not allowed to overdraw the account, the credit will be zero. The credit is a constant of that particular interaction between the customer and the account that instantiate the roles. Hence, no action of the glue modifies the 'credit' variable.

```
connector Movement(Customer, AnyAccount)
  design Movement
  in     amount : nat; balance : int
  prv    credit : nat
  do     put: true → skip
  []     get: balance - amount ≥ -credit → skip
  channel amount-value put-deposit get-withdraw
  channel amount-value put-credit get-debit balance-balance
end connector
```

Notice that the name of a connector is the name of its glue, and that the i-th channel describes the bindings between the glue and the i-th role.

An n-ary connector is applied to n components by defining which component refines which role in which way. Usually, general-purpose connectors (e.g., for asynchronous communication or RPC) have also very general roles (i.e., which impose very little restrictions on the components that intantiate them), because such connectors are supposed to be applicable to a wide class of components. This means that several different component designs may refine the same role. Moreover, for a given role-component pair there may be several different possible refinements. However, for a particular architecture, several of these possibilities may not make sense. Normally, each role is to be refined by only a few component designs and each one of them has to refine the role in only one way. Hence, we require the architect to specify just those refinements that are possible. Identity and inclusion refinement morphisms are always allowed.

For our example, 'AnyAccount' can be refined in two ways.

```
refinement AN: AnyAccount → NormalAccount
value/amount, balance/balance, credit/deposit, debit/withdraw
end refinement

refinement AS: AnyAccount → SalaryAccount
AN; inclusion
end refinement
```

Moreover, there is the identity between the 'Customer' role and the component design.

3.2 Constraints

The components, connectors, and refinements specified by the architecture only provide the vocabulary to write configurations. In other words, they only impose minimal, syntactic restrictions on the class of configurations for an intended software system.

To describe more complex constraints we use a first-order language where variables are typed by designs or connectors. Such variables range over the nodes of the current run-time configuration, in the case of a connector the referred node being the glue. The *connector predicate*

$$\mathit{ConnectorVar}([\mathit{RefinementName}_1 \rightarrow]\ \mathit{ComponentVar}_1, \ldots)$$

is satisfied if the configuration contains a star-shaped subgraph with the node *ConnectorVar* connected to the given component nodes

ComponentVar$_i$ through channels obtained by composing the channels from the glue *ConnectorVar* to the roles with the given refinements from the roles to the components. If it is not relevant how one of the roles is intantiated, both *RefinementName* and *ComponentVar* may be the anonymous variable, which we write as in Prolog, using the underscore character. Moreover, we use the usual notation $[X]$ to describe optional syntactic elements. In this case, the name of the refinement is omitted if it is an identity or an inclusion.

Since uniqueness constraints are quite common (e.g., in a token-ring topology, exactly one node has the token at any time), we also use the unique existential quantifier (∃!).

For example, the following formula states that every customer must be connected to exactly one account and vice versa.

$$
\begin{array}{l}
(\forall\ \text{c:Customer}\ \exists!\ \text{m:Movement m(c, \_)}) \wedge \\
(\forall\ \text{a:NormalAccount}\ \exists!\ \text{m:Movement m(\_, AN} \longrightarrow \text{a))} \wedge \\
(\forall\ \text{a:SalaryAccount}\ \exists!\ \text{m:Movement m(\_, AS} \longrightarrow \text{a))}
\end{array}
$$

In dynamic reconfiguration, there is an interplay between the topology of the run-time configuration and the state of the nodes. Therefore, constraints may also refer to the local variables of the node designs. Because names of design variables are not global, they must be qualified by the name of the node, for which we use a dot notation: *NodeVar.VariableName*. For our example, we need to specify that if the customer has a normal account, the withdrawal credit is zero; if s/he has a salary account, the withdrawal credit cannot exceed the salary.

$$
\begin{array}{l}
(\forall\ \text{a:NormalAccount; m:Movement} \\
\quad \text{m(\_, AN} \longrightarrow \text{a)} \Rightarrow \text{m.credit} = 0) \wedge \\
(\forall\ \text{a:SalaryAccount; m:Movement} \\
\quad \text{m(\_, AS} \longrightarrow \text{a)} \Rightarrow \text{m.credit} \leq \text{a.salary})
\end{array}
$$

Constraints may also use (in)equality predicates over node variables. For our example, we require account numbers to be unique.

$$
\begin{array}{l}
(\forall\ \text{a1, a2 : NormalAccount a1} \neq \text{a2} \Rightarrow \text{a1.number} \neq \text{a2.number}) \wedge \\
(\forall\ \text{a1, a2 : SalaryAccount a1} \neq \text{a2} \Rightarrow \text{a1.number} \neq \text{a2.number}) \wedge \\
(\forall\ \text{n:NormalAccount; s:SalaryAccount n.number} \neq \text{s.number})
\end{array}
$$

3.3 Configuration Variables

In the same way that components have variables to describe their current state, it is desirable to have variables to store information about the current run-time configuration. Such *configuration variables* may be typed over the available data sorts, component designs, and connector (i.e., glue) designs. Configuration variables can be used in constraints and are only updated by reconfiguration commands. We achieve a clean separation between computation and (re)configuration because design actions can only use the variables declared in the same design, and because no kind of binding between configuration variables and design variables is allowed.

As an example, consider a token ring architecture, in which each component ‘RingComp’ has a boolean local variable ‘hasToken’ that is true only while the component holds the token. The reconfiguration commands to be presented in Section 4.1 allow one to find the node that currently holds the token, but such an operation is costly, being linear in the size of the ring. It is more efficient to have a configuration variable ‘holder:RingComp’ that refers to that node, and that is updated whenever the token is transferred to another node. The semantics of the variable is given by the constraint

$$\text{holder.hasToken} \wedge \exists!\ \text{x:RingComp x.hasToken}$$

which states that only the node referred by ‘holder’ has the token.

As for our running example, we use two counters: one for the number of accounts (which is the same as counting the customers), and another for the number of salary accounts. Normally, a person has her or his salary account in the bank s/he likes most. Therefore, the ratio between the two counters is an indication of the percentage of customers that have this bank as their prime choice.

3.4 Architectures

Putting all together, an architecture defines: the designs that may be used as components (and roles) and the refinement relationships between them; the designs that may be used only for roles, the connectors, and the refinement morphisms for each role; the configuration variables; and a constraint on the possible configurations.

The architecture for our example illustrates the concrete syntax:

```
architecture Bank
components
  design Customer ...
  design NormalAccount ...
  design SalaryAccount ...
connectors
  design AnyAccount ...
  connector Movement(Customer, AnyAccount) ...
variables allAccounts, salAccounts : nat
constraint C
end architecture
```

where C is the conjuntion of the constraints given in Section 3.2. Notice that this architecture has no refinements other than the implicit identities and inclusions.

Within the `components` and `connectors` sections, the definitions of designs, refinements, and connectors may appear in any order, as long as a name is not used before it is defined.

A (run-time) configuration is said to be an *instance* of a given architecture A if:

- all nodes are labelled either with a component or a glue of A;
- each arc connects a component to the glue of a connector;
- each node labelled with the glue of a connector C has exactly as many arcs as the arity of C defined in A, and the i-th arc is labelled by a channel obtained from composing the channel of the glue to the i-th role R_i with one of the possible refinement morphisms between R_i and the component on the other end of the arc.

The second condition prevents two components from being directly connected through a channel, i.e., interactions are only established through connectors. The third condition ensures that each connector was properly applied to components. Notice that the roles only serve to constrain how a connector can be applied; they do not appear in the configuration.

4. RECONFIGURATION

The only possible changes to a configuration are creation and removal of nodes (whether they are components or glues). These basic commands can be combined into more complex changes using sequencing, choice, and iteration operators. Changes can be grouped into scripts, which correspond to procedures in Pascal-like languages. Besides the global configuration variables declared at the architectural level, scripts may declare local configuration variables, and may have configuration variables as parameters. These variables are changed by the scripts and their values are used to compute the new configuration. There is also a command to query the current configuration, returning all tuples of nodes that match some conditions. These nodes can then be processed by the reconfiguration commands.

The semantics of basic reconfiguration commands is given by conditional graph productions over the categorical diagram representing the current run-time configuration. The reconfiguration interpreter executes these productions in the order specified by the scripts and composite commands.

4.1 Commands

As was the case with constraints on the architecture, the reconfiguration language to be presented next makes heavy use of *node references*, which are variables that are typed by designs. A node reference is *undefined* if it does not refer to any node of the current run-time configuration. In the following description of the language, *Node* stands for such a variable. The state of any node can be accessed using a dotted notation: *Node.Var* refers to the local variable *Var* of node *Node*. This makes it possible to write arbitrary expressions *Exp* involving the operations of the data types used by CommUnity and the values of the local variables of the nodes of the current configuration. Of course, *Exp* is undefined if at least one of the node references it uses is undefined.

The examples in this subsection assume the existence of the following declarations:

n : NormalAccount; s : SalaryAccount; c : Customer; number : nat

4.1.1 Component Creation

The command to create a component node is

[*Node* :=]
`create` *Design* `with` l_1 := Exp_1 ‖ l_2 := Exp_2 . . .

where *Node* must be declared of type *Design* and l_i are the local variables of *Design*. All local variables must be assigned to, in order to correctly initialise the new component. This reconfiguration command does nothing if some expression Exp_i is undefined. The reference to the created component is stored in *Node*, if given. The names occurring in Exp_i are resolved in the context in which this command occurs, not in the context of *Design*. This makes it possible to use script parameters that have the same names as the local variables of designs, to improve readability of the script.

For example, consider the following command:

n := `create` NormalAccount `with`
    balance := n.balance ‖ avgbal := 0 ‖ number := number

This creates a new account, transferring the money from an existing account 'n'. The number of the new account is given by the value of the variable declared above. After the command, 'n' refers to the new account. If 'n' is undefined, no component is created (because it is impossible to initialise it), and hence 'n' remains undefined.

4.1.2 Component Refinement

A common reconfiguration is to update a component, e.g., to add new functionality, eliminate bugs, or improve efficiency. In our framework, this is achieved through replacement of a component by a refinement (other than the identity) of it. The syntax is

[$Node_2$:=]
`create` $Design_2$ `as` [*Refinement*(]Node_1[])]
`with` l_1 := Exp_1 ‖ . . .

where $Node_2$ is of type $Design_2$ and $Node_1$'s type is some $Design_1$.

This command removes $Node_1$ and replaces it by a new node $Node_2$. For any glue to which $Node_1$ was connected, through some channel c, the new node becomes connected to the same glue through a channel that is the composition of c with *Refinement*. Only the new local variables l_i—i.e., those of $Design_2$ that do not correspond to any variable of $Design_1$—are assigned to; the rest of the state of $Node_2$ is transferred from $Node_1$.

The command does nothing if $Node_1$ or some Exp_i is undefined, otherwise $Node_1$ becomes undefined after the execution of the command. If the parentheses and the refinement name are omitted, then $Design_1$ and $Design_2$ must be different and the refinement is an inclusion.

For example, the command

s := `create` SalaryAccount `as` n `with` salary := 100000

replaces the normal account 'n' by a salary account with the same balance, average balance, and account number as 'n'. Since design 'SalaryAccount' only adds one variable to design 'NormalAccount', only that variable has to be explicitly initialised.

4.1.3 Connector Creation

The command to create a connector has a slightly different syntax:

[*Node* :=]
`apply` *Connector* ([$Refinement_1$→]$Node_1$, . . .)
`with` l_1 := Exp_1 ‖ l_2 := Exp_2 . . .

where l_i are the local variables of the glue of the *Connector*. This command applies the connector to the given components, creating a node typed by the glue, and linking it to the given components $Node_i$ through channels that are obtained by composition of the channels declared in the connector definition with the refinements given by the command. This composition can be calculated at compile time since all information is available. If the refinement name and arrow are absent, this indicates an identity or inclusion morphism.

This command does nothing if some $Node_i$ or Exp_j is undefined, or if the creation of the connector would make output variables to be shared, which would violate the conditions on configurations.

For example, the command

`apply` Movement(c, AS → s) `with` credit := s.salary

connects the salary account created above with a customer who has chosen the maximum allowed credit, namely her/his salary. In this example the reference to the new node is not stored in any variable.

4.1.4 Removal

The command to remove a node is `remove` *Node*. After this command, *Node* is undefined. As usual, the command does nothing if *Node* was already undefined. If *Node* refers to a glue, then the node and all its attached channels are removed. If *Node* refers to a component, then it is removed only if it is attached to no channels, because otherwise the resulting configuration would not be an instance of the architecture anymore, since the glues connected to *Node* would have an arity less than prescribed by the architectural definition.

For example, the command '`remove` c' (or '`remove` s') does not change the configuration if it comes right after the connector creation command given above, because customer 'c' (resp. account 's') is connected, namely to salary account 's' (resp. customer 'c').

4.1.5 Query

The query expression

`match` {$Decl_1$ [| [$Decl_2$] [*Configuration*]
[`with` *Condition*]]}

returns all tuples of nodes $Decl_1$ for which there exist nodes $Decl_2$ such that the current run-time configuration includes the sub-configuration *Configuration* in a state where *Condition* holds.

Configuration is just a set of (in)equalities or connector predicates (as defined in Section 3.2) over previously declared node variables, in particular those in $Decl_1$ and $Decl_2$. The scope of those two declaration sequences is just the query expression. *Condition* is a boolean expression over the operations provided by the available data types, i.e., it may not contain connector predicates or (in)equalities over node references. In other words, *Condition* tests the state of the current configuration, while *Configuration* tests its topology.

The `with` part can be omitted if the condition is a tautology, and $Decl_2$ and *Configuration* can also be omitted when not necessary, e.g., if we wish just to find in the current run-time configuration all nodes given in $Decl_1$.

For the query expression to be possible, the reconfiguration language must include record types 'record($Field_1$: $Type_1$; $Field_2$: $Type_2$; ...)' and list types 'list(*Type*)' with the usual operations 'head', 'tail', 'cons', 'append', and the empty list constant ∅. The type of the query expression is then 'list(record($Decl_1$))'.

By storing the node tuples that match the query in a list, they become amenable to further manipulation, like iteration, sorting, filtering according to other criteria, counting, etc. Moreover, computing once the nodes on which we want to further operate, eliminates the need to use dummy variables to make sure that no operation is performed twice on the same tuple of nodes.

For example, the expression

`match` {c:Customer | a:SalaryAccount; m:Movement m(c, AS → a)
`with` m.credit = 0}

selects all customers that have a salary account but have chosen not to have any credit. The resulting list can be used for different purposes, e.g., just counting how many customers are in that situation, or transforming those accounts automatically into normal accounts. The latter involves removing the connector and the salary account, creating a normal account with the same balance and number, and then connecting it to the customer. Therefore, a better expression would be

`match` {c:Customer; a:SalaryAccount; m:Movement | m(c, AS → a)
`with` m.credit = 0}

because it would also return the references to the accounts and movements necessary for the ensuing transformations.

As another example, `match` {a:SalaryAccount | a ≠ s} returns all salary accounts except the one referred by variable 's'. If 's' is undefined, so is the configuration condition, and therefore the query returns the empty list.

4.1.6 Assignment

The assignment *Var* := *Exp* updates the configuration variable *Var*, if *Exp* is not undefined. Notice that *Var* is not a local variable of some node, because reconfiguration does not change the state of existing nodes; it only initialises new nodes and removes nodes.

For example, assuming the existence of a function to calculate the length of a list, the assignment

number := length(append(`match` {a:NormalAccount},
`match` {a:SalaryAccount})

would store in 'number' how many accounts the bank currently has.

4.1.7 Composite Commands

The basic commands just presented can be combined into composite commands using the sequencing operator ';', the conditional command

`if` *Condition* `then` *Command* [`else` *Command*] `end if`

and the iteration command

`while` *Condition* `loop` *Command* `end loop`

where *Command* is a basic or composite command, and *Condition* is as explained in Section 4.1.5.

Since iterating through a list of nodes is so frequent, we introduce the following abbreviation:

`for` *Var* `in` *Expression* `loop` *Command* `end loop`

Expression is a list, *Var* must be of the type of the list elements, and this command expands into

```
NewVar := Expression;
while NewVar $\neq \emptyset$ loop
   Var := head(NewVar); NewVar := tail(NewVar);
   Command
end loop
```

For example, to remove all unconnected customers, the command is simply

```
for i in match {c:Customer} loop remove i.c end loop
```

with ‘i : record(c:Customer)’. In fact, it is guaranteed that `remove` does nothing if ‘c’ is connected.

4.2 Scripts

A script is a command that has a name so that it can be called from other scripts. A script may have input and output parameters, and private variables, with concrete syntax as for CommUnity designs (see examples below). Scripts may be recursive (e.g., this is useful to process tree topologies) and nested. Only top-level scripts can be called directly by the user. The coordination between computation and reconfiguration is achieved by a reconfiguration interpreter that constantly executes the following loop:

1. Execute one computation step on the run-time configuration.
2. Let the user execute one of the scripts, if s/he wishes.
3. If there is a script called ‘Main’, execute it.
4. Go to step 1.

Step 2 caters for ad-hoc reconfiguration and step 3 handles programmed reconfiguration. By interleaving computation and reconfiguration (as also done in [13]) one can be assured that changes to the configuration will be triggered by changes to the state as soon as needed. Furthermore, the changes performed in step 2 and 3 are committed to the architecture only if, after execution of the called script, the constraints on the architecture are still valid.

Notice that users may only call scripts; they may not write arbitrary commands. This is to prevent them from making changes that would invalidate the architectural invariants. We assume that only system administrators have the necessary overall knowledge of the system to write scripts. Ad-hoc reconfiguration is also coped by letting the set of available scripts change during system life-time. The administrator may hence add, remove or replace scripts at any time.

We now present the scripts for our bank example. We begin with the main script. It is responsible for transforming salary accounts automatically into normal accounts if the average balance is negative, and thereby updates the ‘salAccounts’ counter.

```
script Main
prv  n : NormalAccount; i : record(m:Movement; c:Customer)
        s : record(a:SalaryAccount)
for s in match {a:SalaryAccount | with a.avgbal $<$ 0}
loop
    i := head(match {m:Movement; c:Customer | m(c, AS $\rightarrow$ s.a)});
    n := create NormalAccount
          with balance := s.a.balance
          || avgbal := s.a.avgbal || number := s.a.number;
    remove i.m;
    remove s.a; salAccounts := salAccounts - 1;
    apply Movement(i.c, AN $\rightarrow$ n) with credit := 0
end loop
end script
```

Notice how the state of the salary account is transferred to the normal account.

The next script creates a normal account, with a given number. It also creates a client and connects it to the account. The script does nothing if an account with the given number already exists.

```
script CreateNormal
in    number : nat
out  n : NormalAccount
prv  c : Customer
if match {a:NormalAccount | with a.number = number} = $\emptyset$
    $\wedge$ match {a:SalaryAccount | with a.number = number} = $\emptyset$
then
    n := create NormalAccount
          with balance := 0 || avgbal := 0 || number := number;
    c := create Customer with value := 0 || ready := false;
    apply Movement(c, AN $\rightarrow$ n) with credit := 0;
    allAccounts := allAccounts + 1
end if
end script
```

The third script creates a salary account, given the salary, the credit the customer wishes, and the account number. The script first checks that the credit asked for is less than the salary. Next, it proceeds according to the three possible cases: if a salary account with that number already exists, nothing is done; if a normal account with that number exists, it is refined into a salary account, and the connector is replaced; otherwise, the salary account has to be created from scratch. Just to illustrate calling of scripts, this is done by creating a normal account and then proceeding as in the second case.

```
script CreateSalary
in    salary, credit, number : nat
out  s : SalaryAccount
prv  n : NormalAccount; l : list(record(n: NormalAccount))
        i : record(m:Movement; c:Customer);
if credit $\leq$ salary $\wedge$
    match {s:SalaryAccount | with s.number = number} = $\emptyset$
then
    l := match {n:NormalAccount | with n.number = number};
    if l = $\emptyset$ then CreateNormal(number, n)
             else n := head(l).n
    end if;
    i := head(match {m:Movement c:Customer | m(c, AN $\rightarrow$ n)});
    remove i.m;
    s := create SalaryAccount as n with salary := salary;
    create Movement(i.c, AS $\rightarrow$ s) with credit := credit;
    salAccounts := salAccounts + 1
end if
end script
```

4.3 Semantics

There are two kinds of commands: the basic commands perform the actual reconfiguration, while the composite commands and scripts only control the flow of execution. The semantics we provide to the language consists in translating the basic commands into the graph rewrite rules (also called *graph productions*) defined in [29, 30]. The composite commands then just instruct the graph rewriting machine in which order the rules are to be executed. We do not present the complete translation process, nor do we provide formal definitions. We just sketch the main idea and provide concrete examples. A report with further details is in preparation.

The compilation of the reconfiguration commands into productions proceeds as follows. First, each component and glue design is compiled into a design that adds a new private variable ‘*NI*: nat’, such that *NI* is a name that does not occur in any design given in the architecture. The purpose of the variable is to hold a unique node identifier. For our example, we assume *NI* is ‘node’.

Second, the compiler generates *configuration designs* that have only private variables. Each design corresponds to one lexical scoping level (i.e., to one reconfiguration script), and the private variables correspond to the configuration variables that are visible in that level. This means that all those designs include the global configuration variables declared with the architecture. If a configuration variable is a node reference (e.g., c:Customer) then it will be compiled into a variable of type ‘nat’ (e.g., c:nat). That variable will hold zero if the node reference is undefined, otherwise it will hold the value of the *NI* variable of the referred node. Moreover, all those configuration designs have a variable ‘*C*: nat’ which counts how many nodes have been globally created, so that each newly created node can get a unique identifier. The name *C* must of course be different from the names of all configuration variables. For our example we assume *C* is ‘nodes’. The actual names of the designs must be chosen by the compiler so that it does not conflict with any other design specified in the architecture. For our example, we assume those designs are named ‘C_*S*’ with *S* ranging over the script names.

A basic command in script s is translated into a set of graph productions in which the left-hand side always includes ‘C_s’ and any other nodes that are referred by the expressions ocurring in the command. The right hand side includes also those same nodes, but ‘C_s’ is updated with the new values for the configuration variables and for the node counter. The right-hand side also includes any newly created component or glue and omits any removed component or glue.

As an example, we show in Figure 1 the graph production corresponding to the component creation command given in the ‘Main’ script in Section 4.2. Each node is labelled with the design name and with design variable/logical value pairs. According to the definitions in [29, 30], the production can only be applied if there is a substitution for the logical variables $vnodes, vn, bal$, etc., such that the left hand side graph can be mapped to a subgraph of the current run-time configuration. Notice that if the node reference ‘s.a’ is undefined (i.e., $sa = 0$), then the production cannot be applied (because ‘node’ is always positive), and the configuration does not change, as required by the description of `create` in Section 4.1.1. Notice also how *C_Main* is updated and how the new node is initialised.

Our next example, in Figure 2, is the translation of the connector removal command of script ‘Main’. Notice that one rule is generated for each possible instantiation of the roles. In this case, there are only two possibilities: the first role is instantiated with a customer and the second one is instantiated with a normal or salary account. In the particular context of the execution of script ‘Main’, by the time one of these rules is executed, one has $vc = cust$. The channel bindings have been omitted from the figure due to space limitations.

The command to replace a component is the most complex one, because one cannot know *a priori* to which connectors the component to be replaced is attached. Therefore the command is compiled into a set of productions that do the replacement in three phases: the first introduces the new component, the second relinks all connectors to the new component, and the last phase removes the original component. For the second phase, there is one production for each role of each connector that the node to be replaced may instantiate. For example, if there were two connectors $C_1(R_1, R_2)$ and $C_2(R_4, R_5, R_6)$, and a node n_1 of type N_1 were to be replaced by a node n_2 of type N_2, with refinements from R_1, R_4 and R_6 to N_1, then there would be three productions, one replacing the first channel of C_1 to n_1 by a channel from C_2 to n_2, another production for the first channel of C_2 and the last production for the third channel of C_2. The set of productions generated for this second phase of the replacement is to be applied until no left-hand side can be matched to the current configuration. At that point, the node to be replaced has no connector attached and the single rule for the last phase can be applied: it simply removes the node, updating the node reference given in the command.

As a concrete example, let us take the component refinement command of ‘CreateSalary’. The semantics of this command is given by three productions (Figure 3). The first introduces a new salary account, transfers the state from ‘n’, initialises the ‘salary’ local variable, and updates the node reference ‘s’. The second production rebinds any ‘Movement’ connector that might be attached to ‘n’ to the new node ‘s’. The rebinding is done by moving the channel between the glue and the old node to the new node. The last rule removes the node referred by ‘n’, which becomes undefined (i.e., zero).

5. CONCLUDING REMARKS

Dynamic reconfiguration is an important topic for an increasing number of software systems that must be continuously available while coping with all kinds of changes. However, to our knowledge, there is no work that tackles this problem from the Software Architecture point of view and provides a formally based language that integrates the three aspects argued for in [26]: architectural description, constraint, and modification. This paper is a first step in that direction.

We have provided an approach that is both heterogeneous (for pragmatic reasons) and uniform (for formal reasons). It is heterogeneous because it provides an dedicated, separate sub-language for each aspect: a program design language for computation, a declarative language for constraints, and an operational language for reconfiguration. It is uniform because it uses Category Theory as a semantic foundation both for configurations (taken as categorical diagrams) and reconfiguration (achieved through algebraic graph

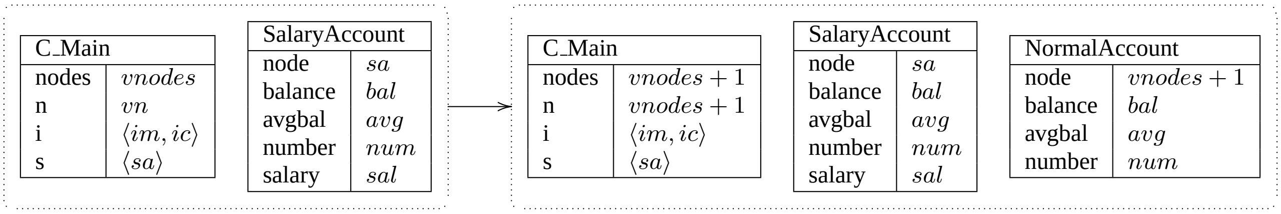

Figure 1: Semantics of ‘n := `create` NormalAccount `with` balance := s.a.balance || avgbal := s.a.avgbal || number := s.a.number’

rewriting techniques [6]).

The approach also provides a strict separation between computation and (re)configuration, while keeping them explicitly related. This was already done at the formal level in our previous papers [29, 30], and the language presented herein complies with that principle: the components do not have access nor can change the configuration variables or call scripts, and the reconfiguration scripts have access but cannot change the state of components. Notice that replacing a component by another one of the same design but with a different state is not what we mean by state change, because there are actually two components involved: the original one is removed and a new one is created.

The main goal of our language is to provide high-level constructs that are suited to the architectural level of description of a system. In particular, interactions are created and removed at the level of connectors, hence guaranteeing that configurations are always instances of the architecture. This contrasts with other approaches which, like the Armani ADL [24], force the designer to write explicit constraints to prevent dangling roles, or, like distributed systems approaches, require reconfiguration scripts to connect the components port by port.

Because our emphasis was on obtaining an integrated language that covers the three aspects (description, constraints, modification), we are aware that other approaches handle some individual aspects better, but in turn they have to sacrifice formality or breadth. For example, Armani allows more complex architectures and constraints, Gerel [7] provides more expressive queries, and Goudarzi [23] takes full advantage of the Java language to program reconfigurations.

Therefore, in future work, we want to further investigate which fundamental constructs (not syntactic sugar!) are needed for the various aspects, and how they can be formally grounded on our algebraic framework. Practical feedback from such research will be gathered by incorporating such reconfiguration primitives into a tool [12] being built to construct and manage coordination contracts—which are akin to connectors—among components implementing core business functionalities [3].

Two other issues are also of concern to us. First, we would like to be able to prove whether a given script maintains the architectural invariants. For that purpose we intend to adapt the logic developped in [10]. Second, we want to extend this work to allow the architecture itself to evolve. This means that during some period the configuration is not an instance of any architecture, neither the original nor the new one. This poses interesting problems both for the language design (because the script will be typed by two architectural descriptions) and for the verification of the correctness of such a reconfiguration process. We will look into [14, 8] and see how it can be adapted to our framework.

Acknowledgments

We thank Luís Andrade and the anonymous reviewers for their comments.

6. 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] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM TOSEM*, 6(3):213–249, July 1997.

[3] L. Andrade and J. L. Fiadeiro. Coordination technologies for managing information system evolution. In *Proc. CAiSE’01*, volume 2068 of *LNCS*, pages 374–387. Springer-Verlag, 2001.

[4] 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.

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

[6] 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.

[7] 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.

[8] G. Engels and R. Heckel. Graph transformation as unifying formal framework for system modeling and model evolution. In *Proc. of ICALP*, volume 1853 of *LNCS*, pages 127–150. Springer-Verlag, 2000.

[9] J. L. Fiadeiro and A. Lopes. Semantics of architectural connectors. In *Proceedings of TAPSOFT’97*, volume 1214 of *LNCS*, pages 505–519. Springer-Verlag, 1997.

[10] J. L. Fiadeiro, N. Martí-Oliet, T. Maibaum, J. Meseguer, and I. Pita. Towards a verification logic for rewriting logic. In *Recent Trends in Algebraic Development Techniques*, number 1827 in LNCS, pages 438–458. Springer-Verlag, 2000.

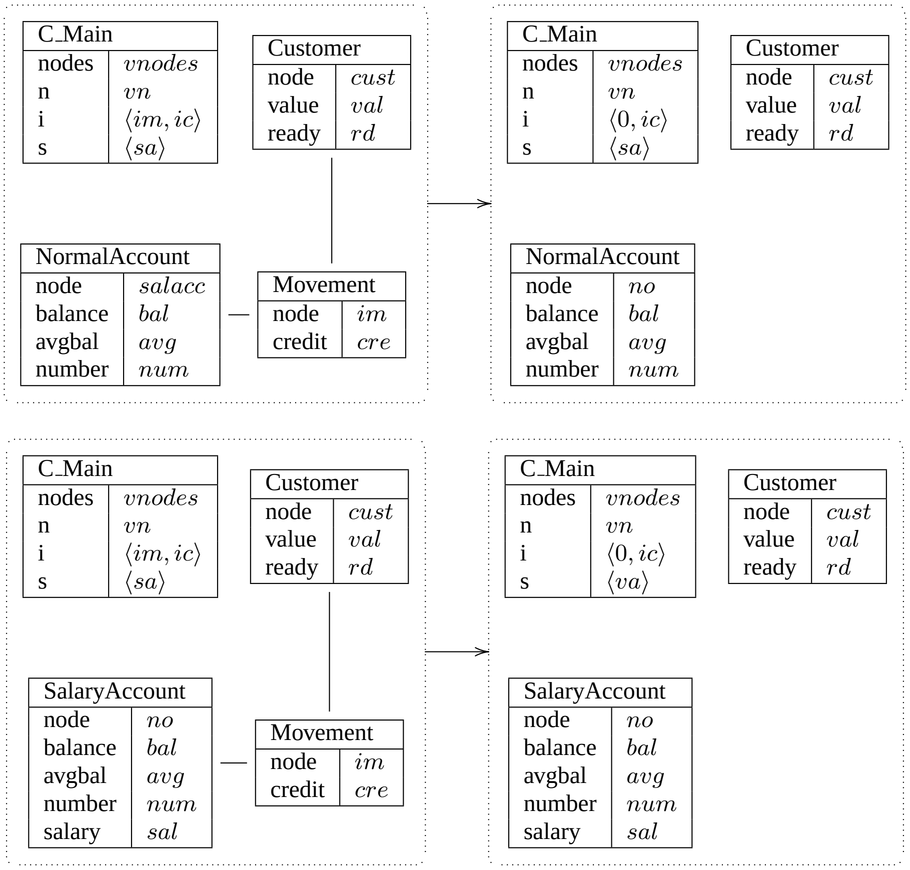

Figure 2: Semantics of 'remove i.m'

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

[12] J. Gouveia, G. Koutsoukos, L. Andrade, and J. L. Fiadeiro. Tool support for coordination-based software evolution. In *Proc. TOOLS 38*, pages 184–196. IEEE Computer Society Press, 2001.

[13] 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.

[14] D. Hirsch, P. Inverardi, and U. Montanari. Reconfiguration of software architecture styles with name mobility. In *Coordination Languages and Models*, volume 1906 of *LNCS*, pages 148–163. Springer-Verlag, 2000.

[15] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, Nov. 1990.

[16] A. Lopes. *Não-determinismo e Composicionalidade na Especificação de Sistemas Reactivos*. PhD thesis, Universidade de Lisboa, Jan. 1999.

[17] A. Lopes and J. L. Fiadeiro. Using explicit state to describe architectures. In *Proceedings of Fundamental Approaches to Software Engineering*, number 1577 in LNCS, pages 144–160. Springer-Verlag, 1999.

[18] 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.

[19] J. Magee, J. Kramer, and D. Giannakopoulou. Behaviour analysis of software architectures. In *Software Architecture*, pages 35–50. Kluwer Academic Publishers, 1999.

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

[21] N. R. Mehta, N. Medvidovic, and S. Phadke. Towards a taxonomy of software connectors. In *Proc. of the 22nd Intl. Conf. on Software Engineering*, pages 178–187. ACM Press, 2000.

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

[23] K. Moazami-Goudarzi. *Consistency Preserving Dynamic Reconfiguration of Distributed Systems*. PhD thesis, Imperial College London, Mar. 1999.

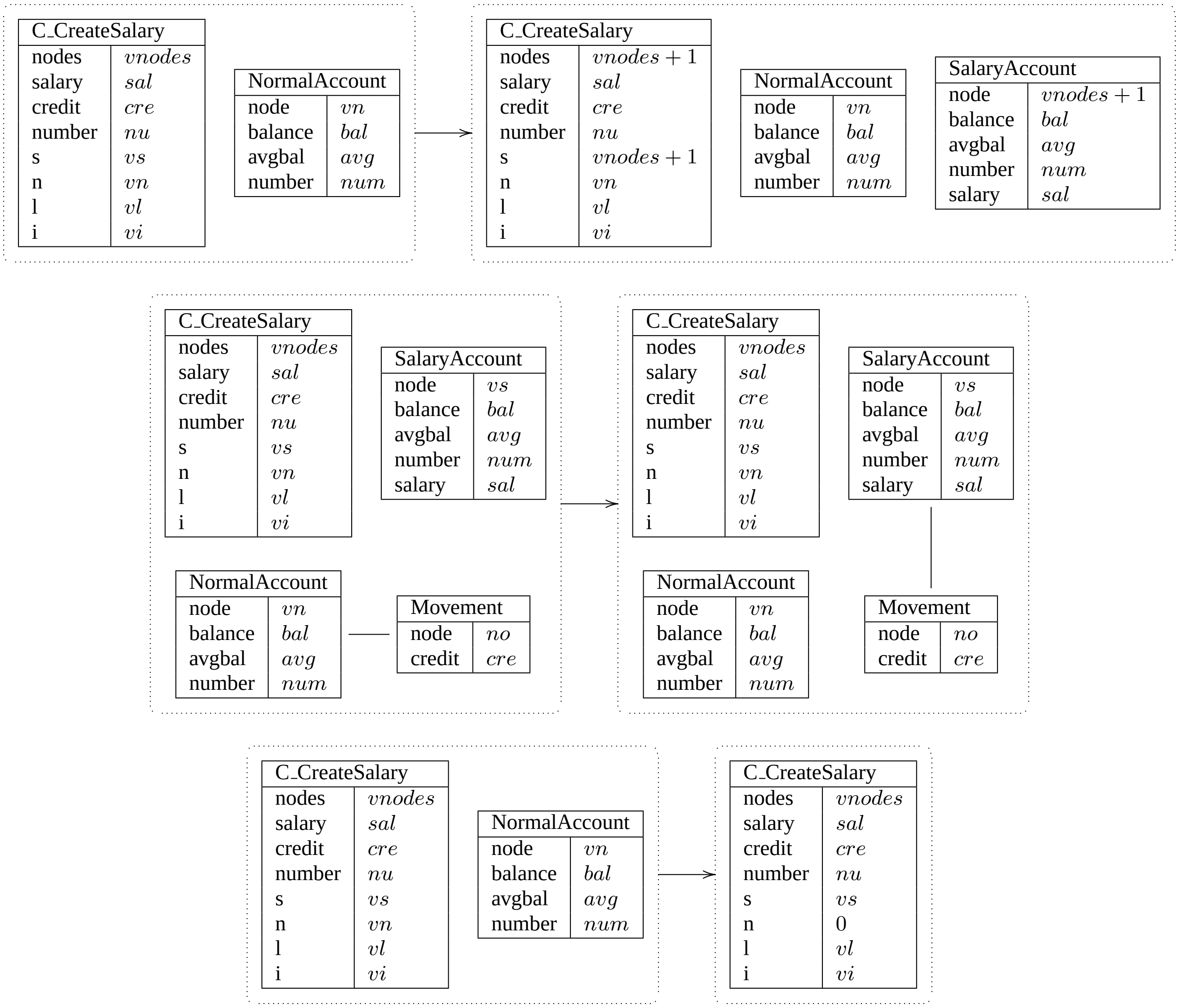

Figure 3: Semantics of 's := create SalaryAccount as n with salary := salary'

[24] R. T. Monroe. Capturing software architecture design expertise with Armani. Technical Report CMU-CS-98-163, School of Computer Science, Carnegie Mellon University, Oct. 1998.

[25] R. T. Monroe, D. Garlan, and D. Wile. *Acme StrawManual*, Nov. 1997.

[26] P. Oreizy. Issues in the runtime modification of software architectures. Technical Report UCI-ICS-TR-96-35, Department of Information and Computer Science, University of California, Irvine, Aug. 1996.

[27] P. Rodgers. Constructs for programming with graph rewrites. In *Joint APPLIGRAPH and GETGRATS Workshop on Graph Transformation Systems*, number 2000-2 in Forschungsberichte des Fachbereichs Informatik, pages 59–66. Technische Universität Berlin, 2000.

[28] 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.

[29] 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.

[30] M. Wermelinger and J. L. Fiadeiro. A graph transformation approach to run-time software architecture reconfiguration. *Science of Computer Programming*, 2001. To appear.