Synchronized Hyperedge Replacement for Heterogeneous Systems*

Ivan Lanese and Emilio Tuosto

Dipartimento di Informatica, Largo Bruno Pontecorvo 3,
56127 Pisa – Italy

Abstract. We present a framework for modelling heterogeneous distributed systems using graph transformations in the Synchronized Hyperedge Replacement approach, which describes complex evolutions by synchronizing local rules. In order to deal with heterogeneity, we consider different synchronization algebras for different communication channels. The main technical point is the interaction between synchronization algebras and name mobility in the π-calculus style. The power of our approach is shown through a few examples.

1 Introduction

Nowadays, Internet is becoming more and more integrated with *non* IP-based networks and the so-called *overlay networks*, which combine different kinds of networks (e.g., ATM, wireless etc.), are beginning to appear in the scene. Overlay networks are changing the nature of Internet, indeed new protocols and communication policies are required in order to allow applications (such as web services) to interact across different kinds of networks.

Doubtless, applications should consider these changes of perspective in order to take advantage of the new technological possibilities offered by overlay networks. Indeed, not only this evolution of Internet has impact on the communication infrastructure, but it also influences the level of the applications and the middlewares they rely on. For instance, in some cases it would help to have point-to-point communications while in others broadcast is preferable. Of course, this kind of situation may be present also in the usual practice of concurrent programming. A typical example is when a server first acquires data over which it computes and then must send the results to several waiting clients. Classical languages for concurrent/distributed programming offer a limited number of communication policies while models usually restrict on a single synchronization mechanism. For example, the π-calculus [13, 15] adopts point-to-point communication, hence, broadcast communications, if desired, must be encoded. Classical formalisms, however, cannot uniformly deal with dynamical/unpredictable variations of the synchronization policies.

* I. Lanese has been supported by EU-FET project **AGILE** IST-2001-32747. E. Tuosto has been supported by EU-FET project **PROFUNDIS** IST-2001-33100.

J.-M. Jacquet and G.P. Picco (Eds.): COORDINATION 2005, LNCS 3454, pp. 220–235, 2005.
© Springer-Verlag Berlin Heidelberg 2005

The *Synchronized Hyperedge Replacement* approach [1, 3] (SHR, for short) is a uniform graph transformation framework for dealing with many facets of wide area network applications [7, 16, 8, 2, 6, 11, 5]. Systems are hypergraphs, namely graphs where each hyperedge connects an arbitrary number of nodes, computations correspond to rewrite hypergraphs by applying *productions* which are rules of the form $p : L(\boldsymbol{x}) \to G$ where $L(\boldsymbol{x})$ is an hyperedge and G an hypergraph. Informally, applying production p to a graph means to replace an instance of L with G in the graph. The replacements are coordinated in SHR through the requirements that p imposes to the attachment nodes of L, namely, in order to replace L with G, it is necessary that the components connected to the attachment nodes of L in the graph synchronize (according to a given synchronization policy) with the requirements imposed by p. Intuitively, this implies that synchronizing corresponds to resolving a distributed constraint satisfaction problem as pointed out in [14]. The SHR approach has the advantage, w.r.t. other graphical frameworks, such as double push-out [4] and bigraphs [10], of allowing a distributed implementation since productions have a local effect and synchronization can be performed using a distributed algorithm.

A first abstraction w.r.t. synchronization mechanisms has been done in [12], where SHR has been equipped with a general notion of *synchronization algebra with mobility* (SAM, for short). Synchronization algebras [17] abstractly define the basic properties of synchronization policies by distilling the axioms capturing them. Therefore, the programmer can define his own synchronization mechanism and exploit SHR for representing systems and their computations. However, the proposal of [12] lacks the possibility of representing different SAMs in a single rewriting system. The main contribution of this work is to solve this problem.

Following [9], the SHR mechanism used here, allows mobility by means of node fusions. Our extension adds to fusion the peculiarity of changing the synchronization policy of a given node. In particular, SAMs form a commutative monoid which is programmer-defined. Moreover, whenever x is the node resulting from merging y and z, the SAM associated to x is obtained by composing the SAMs associated to y and z via the monoidal operation. Even though this might have no counterpart at the level of the communication infrastructure, this mechanism may result extremely useful at the level of the applications. For instance, the synchronization policy of a new channel can be dynamically determined as the composition of different constraints imposed by interacting components.

Structure of the Paper. §2 gives some background on graphs. §3 discusses synchronization algebras with mobility and labelled graphs. The following sections §4 and §5 present productions and transitions for SHR, respectively via an informal example and with rigorous mathematical definitions. Another example is presented in §6. Finally, §7 presents conclusions and traces for future work.

2 Background

Before describing hypergraphs, some notations are given.

Given a vector $\boldsymbol{v}$, $|\boldsymbol{v}|$ is its length and $\boldsymbol{v}[i]$ is its i-th element.

We denote with $\mathrm{mgu}(E)$ an idempotent substitution resulting from computing the most general unifier on the set of equations E. The transitive closure of E is E^+.

We use $\uplus$ for disjoint set-union and, in $A \uplus B$, $[1, n]$ (resp. $[2, n]$) is the element that corresponds to $n \in A$ (resp. $n \in B$). We denote with Int_n the set $\{1, \ldots, n\}$ (where $Int_0 \stackrel{\text{def}}{=} \emptyset$) while id_n is the identity function on it. Given two functions $f : A \to C$ and $g : B \to D$ we denote with $[f, g] : A \uplus B \to C \uplus D$ the function that applies f to the elements of A and g to the ones of B. The standard composition of functions is denoted by $\circ$. Given a function f, $f|_S$ (resp. $f|_{\setminus S}$) is the function obtained by restricting f to S (resp. to $\mathrm{dom}(f) \setminus S$) while $\mathrm{merge}(f)$ yields the set of equalities $\{x = y | f(x) = f(y)\}$. Finally, when set operations (e.g., $\cup$) are used on functions, it is implicitly assumed that these are represented as sets of pairs.

We want to model systems using hypergraphs, a generalization of graphs where hyperedges may connect any number of nodes. For simplicity, we use graph (resp. edge) instead of hypergraph (resp. hyperedge). We assume a set LE of labels and a function $\mathrm{rank} : LE \to \omega$ that assigns a rank to each $L \in LE$. An edge labelled by L is an atomic item with $\mathrm{rank}(L)$ ordered tentacles. A set of nodes, together with a set of edges, forms a graph if each edge is connected, by its tentacles, to its attachment nodes. A graph is connected to its environment by an interface which is a subset of its nodes. Nodes in the interface are called *free* nodes, while other nodes are said *bound*. We consider graphs up to isomorphisms that preserve free nodes, labels of edges, and connections between edges and nodes.

We use a textual representation for graphs as (syntactic) judgements which is more suitable for defining transformations [9]. In this representation nodes correspond to names, free nodes to free names and edges to basic terms of the form $L(x_1, \ldots, x_n)$, where x_i are arbitrary names and $L \in LE$ has rank n. The constant *nil* represents a graph without edges, the parallel composition operator $|$ builds large graphs from smaller ones and the ν operator binds nodes.

Definition 1 (Graphs as judgements). *Let $\mathcal{N}$ be a fixed infinite set of names. A* judgement *is a pair of the form $\Gamma \vdash G$ where:*

1. *$\Gamma \subseteq \mathcal{N}$ is a finite set of names (the free nodes of the graph);*
2. *G is a term generated by the grammar*
   $G ::= L(\boldsymbol{x}) \mid G|G \mid \nu\, y\; G \mid nil$
   where $\boldsymbol{x}$ is a vector of names, L is an edge label with $\mathrm{rank}(L) = |\boldsymbol{x}|$ and y is a name.

Table 1. Structural congruence for graph terms

(ax1) $G_1 \vert (G_2 \vert G_3) \equiv (G_1 \vert G_2) \vert G_3$	(ax2) $G_1 \vert G_2 \equiv G_2 \vert G_1$	(ax3) $G \vert nil \equiv G$
(ax4) $\nu\, x\; \nu\, y\; G \equiv \nu\, y\; \nu\, x\; G$	(ax5) $\nu\, x\; G \equiv G$ if $x \notin \mathrm{fn}(G)$	
(ax6) $\nu\, x\; G \equiv \nu\, y\; G\{y/x\}$ if $y \notin \mathrm{fn}(G)$		
(ax7) $\nu\, x\; (G_1 \vert G_2) \equiv (\nu\, x\; G_1) \vert G_2$ if $x \notin \mathrm{fn}(G_2)$		

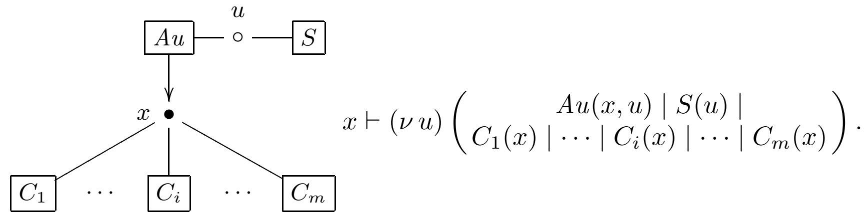
The restriction operator ν *is a binder (similar to the binder of* λ*-calculus). We denote with* $\mathrm{fn}(-)$ *the function that yields the set* $\mathrm{fn}(G)$ *of free names in a term* G*. We demand that* $\mathrm{fn}(G) \subseteq \Gamma$*.*

Graph terms are considered up to the axioms of structural congruence in Table 1: (ax1), (ax2) and (ax3) define respectively the associativity, commutativity and identity over nil for operation $|$. Axioms (ax4) and (ax5) state that nodes of a graph can be hidden only once and in any order. Thanks to axiom (ax4) we can write $\nu\, Z$ where $Z = \{x_1, \ldots, x_n\}$ instead of $\nu\, x_1 \ldots \nu\, x_n$. Axiom (ax6) defines α-conversion of bound names in a graph. Axiom (ax7) defines the interaction between restriction and parallel composition. Note that function $\mathrm{fn}(-)$ is well-defined on equivalence classes. As far as judgements are concerned, we define $\Gamma \vdash G \equiv \Gamma' \vdash G'$ iff $\Gamma = \Gamma'$ and $G \equiv G'$.

Theorem 1 (Soundness of the Representation [7]). *Judgements up to structural congruence are isomorphic to graphs up to isomorphisms.*

In order to explain the formal definitions, we describe here a sample scenario from the Internet realm: it will also be exploited later as a running example.

Some clients $C_1, \ldots, C_m$ can invoke a service offered by a remote server S provided that they are authorized. A possible solution might be to interpose an authority Au between S and the clients. Both S and the clients trust Au, and clients get access to S only after they have been authenticated by Au.



$$x \vdash (\nu\, u) \begin{pmatrix} Au(x, u) \mid S(u) \mid \\ C_1(x) \mid \cdots \mid C_i(x) \mid \cdots \mid C_m(x) \end{pmatrix}.$$

The picture and the judgement above represent clients connected to Au on a "public" node x. The server S is also connected to the authority Au, but this time on a "private" (i.e., restricted) node, graphically represented as an empty bullet. Notice that Au has an arrowed tentacle. In general, hypergraphs are not oriented, here this graphical convention is only used for representing the order of the tentacles in edges having rank $h > 1$. Namely, the arrowed tentacle is the first one while the others are numbered clock-wise.

3 Synchronization Algebras with Mobility

In this section we introduce *synchronization algebras with mobility* (SAM, for short), an extension of synchronization algebras [17] able to deal with name mobility in the style of name-passing calculi. We extend graphs by labelling nodes with SAMs that will be exploited to choose the policies for synchronizing rewriting rules.

Definition 2 (Synchronization Algebra with Mobility). *Let Act be a set of actions containing a distinguished element ϵ and at least an action $a \neq \epsilon$. A SAM on Act, $< N, Act, \mathrm{ar}, \epsilon, Fin, ActCmp >$, is identified by its name N and it includes a function* ar : $Act \rightarrow \omega$ *such that* $\mathrm{ar}(\epsilon) = 0$*, a subset Fin of final actions containing ϵ and a relation ActCmp for action composition. In particular, ActCmp is a set of triples of the form* $(a, b, (c, \mathrm{Mb}))$ *where* $a, b, c \in Act$ *and* Mb *is a partial function from* $Int_{\mathrm{ar}(a)} \uplus Int_{\mathrm{ar}(b)}$ *to* $\{1, 2, \dots\}$ *such that:*

1. $c = \epsilon \Leftrightarrow a = b = \epsilon$*;*
2. Mb *is surjective on* $Int_{\mathrm{ar}(c)}$*;*
3. $(b, a, (c, \mathrm{Mb}')) \in ActCmp$*, with* $\mathrm{Mb}'([1, n]) = \mathrm{Mb}([2, n])$ *and* $\mathrm{Mb}'([2, n]) = \mathrm{Mb}([1, n])$ *for each n;*
4. $(c, d, (e, \mathrm{Mb}')) \in ActCmp \Rightarrow \exists (b, d, (f, \mathrm{Mb}'')) \in ActCmp, (a, f, (e, \mathrm{Mb}''')) \in ActCmp$ *and in that case the composition of* Mb *and* Mb' *gives the same mapping and the same fusions of the composition of* Mb'' *and* Mb'''*.*

The requirements for condition 4 can be written as:

$$\mathrm{Mb}_1 \mid_{Int_{\mathrm{ar}(e)}} = \mathrm{Mb}_2 \mid_{Int_{\mathrm{ar}(e)}}$$
$$(\mathrm{merge}(\mathrm{Mb}) \cup \mathrm{merge}(\mathrm{Mb}_1))^+ = (\mathrm{merge}(\mathrm{Mb}'') \cup \mathrm{merge}(\mathrm{Mb}_2))^+.$$

where $\mathrm{Mb}_1 = \mathrm{Mb}' \circ [\mathrm{Mb} \mid_{Int_{\mathrm{ar}(c)}}, id_{\mathrm{ar}(d)}]$ *and* $\mathrm{Mb}_2 = \mathrm{Mb}''' \circ [id_{\mathrm{ar}(a)}, \mathrm{Mb}'' \mid_{Int_{\mathrm{ar}(f)}}]$.

We define the functions factset*,* factfin *and* factcmp *that given a SAM A compute respectively its set of actions Act, its set of final actions Fin and its composition relation ActCmp.*

Intuitively, $\mathrm{ar}(a)$ is the number of nodes that are communicated with a, ϵ stands for "no-action", Fin contains the set of actions that represent complete synchronizations and thus can be executed also on restricted nodes. Notice that ϵ is always considered complete. Finally, $ActCmp$ defines action synchronization and name communication. More precisely, the triples of the form $(a, b, (c, \mathrm{Mb}))$ define the allowed synchronizations when actions a and b interact: each triple is an allowed behaviour, and if no such triple exists the actions are not compatible, i.e. they cannot synchronize. In particular c is the result of the synchronization and Mb describes how the parameters of a and b are mapped to the parameters of c. If many parameters are mapped to the same position, the corresponding nodes are merged and the resulting node is attached to action c. Only the parameters up to $\mathrm{ar}(c)$ are actually communicated, the others are used for performing additional fusions without showing the result in the final action.

Condition 1, present in normal synchronization algebras, amounts to say that the effect of a synchronization cannot be "no action". Condition 2 guarantees that each node attached to the composed action can be computed, that is it corresponds to a non empty set of nodes from component actions. Finally, conditions 3 and 4 state that action synchronization and mobility patterns are commutative and associative.

We inherit the characterization of mobility from [12] so that general mobility patterns can be modelled. Since this work aims at formalizing heterogeneous systems, we tailor the examples to show the use of multiple SAMs, while maintaining them as simple as possible. For an application of the full generality of the mobility patterns, we refer to [12]. The main difference w.r.t. the SAMs in [12] is that we allow nondeterminism, namely, instead of being fixed, the result of synchronizing two actions is nondeterministically chosen from a set of allowed behaviours specified by the synchronization relation *ActCmp*. This allows to model some more policies, for instance, in a SAM with priorities, when two messages with the same priority interact, one of them is nondeterministically discarded. Also, SAMs are named so that we can distinguish among those describing the same interactions. This can be useful when SAM composition does not depend only on their synchronization policy.

We present here two simple examples of SAMs, namely the one for *Milner synchronization* and the one for *broadcast*. Both of them rely on a mobility pattern that implements message passing, i.e. it merges the corresponding parameters. We define the function message passing $MP_{n,m}$ from $Int_n \uplus Int_m$ to $Int_{\max(n,m)}$ as follows: the element i of both the starting sets (when it exists) is mapped to the element i in the codomain.

When defining SAMs we suppose that *ActCmp* contains just the tuples given explicitly, plus the ones derivable from commutativity.

Example 1 (Milner SAM). Given the set of actions $Act = \{\tau, \epsilon\} \cup \bigcup_{i \in I}\{a_i, \overline{a_i}\}$ where $\mathrm{ar}(\overline{a_i}) = \mathrm{ar}(a_i)$ and $\mathrm{ar}(\tau) = 0$, the Milner SAM over *Act* is as follows:

- $(a, \epsilon, (a, MP_{\mathrm{ar}(a),0})) \in ActCmp$ for each $a \in Act$,
  $(a, \overline{a}, (\tau, MP_{\mathrm{ar}(a),\mathrm{ar}(\overline{a})})) \in ActCmp$ for each $a \in \bigcup_{i \in I}\{a_i\}$;
- $Fin = \{\tau, \epsilon\}$.

Milner synchronization represents message passing *à la* π-calculus, with one process executing input a and the other one executing output $\overline{a}$.

Example 2 (Broadcast SAM). Given the set of actions $Act = \{\epsilon\} \cup \bigcup_{i \in I}\{a_i, \overline{a_i}\}$ where $\mathrm{ar}(\overline{a_i}) = \mathrm{ar}(a_i)$, the broadcast SAM over *Act* is as follows:

- $(a, \overline{a}, (\overline{a}, MP_{\mathrm{ar}(a),\mathrm{ar}(\overline{a})})) \in ActCmp$ for each $a \in \bigcup_{i \in I}\{a_i\}$,
  $(a, a, (a, MP_{\mathrm{ar}(a),\mathrm{ar}(a)})) \in ActCmp$ for each $a \in \bigcup_{i \in I}\{a_i\}$,
  $(\epsilon, \epsilon, (\epsilon, MP_{0,0})) \in ActCmp$;
- $Fin = \{\epsilon\} \cup \bigcup_{i \in I}\{\overline{a_i}\}$.

This SAM implements broadcast, where one process performs an output $\overline{a}$ and *all* the others have to perform an input a. Notice that, if one wants to have

weak broadcast, where some process may not synchronize with the output, it is enough to add the triples $(a, \epsilon, (a, MP_{\mathrm{ar}(a),0}))$ for each $a \in Act$ to $ActCmp$.

We always consider a set $\mathcal{A}\mathfrak{lg}$ containing all the SAMs of interest. This set is assumed to be a commutative monoid w.r.t. an operator $\diamond$ of algebra composition. Names of SAMs are assumed to be unique in $\mathcal{A}\mathfrak{lg}$.

We can now extend graphs by labelling nodes with SAMs. We concentrate on the representation as syntactic judgements.

Definition 3 (Labelled Graphs). *Assuming $A \in \mathcal{A}\mathfrak{lg}$, we define a* labelled graph *as a pair of the form $\Gamma \vdash G$ where:*

1. *Γ is a finite function mapping nodes to SAMs, written as sequence of pairs $n : A$ where $n \in \mathcal{N}$;*
2. *G is a term generated by the grammar*
   $G ::= L(\boldsymbol{x}) \mid G|G \mid \nu\, y : A.G \mid nil$
   where $\boldsymbol{x}$ is a vector of names, L is an edge label with $\mathrm{rank}(L) = |\boldsymbol{x}|$ *and y is a name.*

In $\nu\, y : A.G$, ν binds y in G while recording the label A.

When defining the interfaces, $\Gamma, x : A$ denotes the set obtained by adding $x : A$ to Γ, assuming $x \notin \mathrm{dom}(\Gamma)$ and Γ_1, Γ_2 denotes the union of Γ_1 and Γ_2, assuming $\mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) = \emptyset$.

The relation of structural congruence on graphs is the same that we defined in §2, with α-conversion preserving the labelling SAMs. Now graph isomorphisms also preserve SAMs that label nodes. Theorem 1 holds also with the new definitions.

4 SHR via an Example

Productions are the basic rules of SHR describing how edges can be rewritten. This section gives a flavour of productions and synchronization before the formal definitions which are in the following section. In particular, it highlights how multiple SAMs can model issues of distributed applications in a simpler way w.r.t. previous approaches.

A production takes the form $\boldsymbol{x} : \boldsymbol{A} \vdash L(\boldsymbol{x}) \xrightarrow{\Lambda} \Gamma \vdash G$ where

- $\boldsymbol{x} : \boldsymbol{A}$ abbreviates $x_1 : A_1, \ldots, x_{|\boldsymbol{x}|} : A_{|\boldsymbol{x}|}$ and $\boldsymbol{x} : \boldsymbol{A} \vdash L(\boldsymbol{x})$ is an edge such that all its attachment nodes are distinct,
- $\Gamma \vdash G$ is a labelled graph and
- Λ is a synchronization function mapping nodes in $\boldsymbol{x}$ to pairs $(a, \boldsymbol{y})$ where $a \in Act$ is the performed action and $\boldsymbol{y}$ contains the communicated nodes.

Roughly, such a production states that, in a given graph, an edge labelled L can be replaced by G provided that its attachment nodes $\boldsymbol{x}$ are labelled by $\boldsymbol{A}$ and it can synchronize through actions specified by Λ with the productions of the edges sharing the attachment nodes in $\boldsymbol{x}$.

We present the productions for our running example, i.e., the ones expressing the behaviours of Au, S and a generic client C_i. For the sake of simplicity, we consider the Milner SAM on actions $\{\text{req}\} \cup \bigcup_{i \in Int_m} \{\text{auth}_i\}$ (see Example 1) and call it Mil.

First consider a generic client C_i:

$$x : Mil \vdash C_i(x) \xrightarrow{(x, \overline{\text{auth}_i}, \langle y \rangle)} x : Mil, y : Mil \vdash C_i'(y) \quad (1)$$

$$y : Mil \vdash C_i'(y) \xrightarrow{(y, \overline{\text{req}}, \langle\rangle)} y : Mil \vdash C_i'(y). \quad (2)$$

Production 1 models the authentication phase where C_i is attached to a Mil node and asks Au for the access to the service S. If the authentication takes place, C_i connects to the server through y, the name which will be instantiated with the server "address" during the synchronization with Au. The right-hand-side of production 1 expresses that the client is connected to the server and can make its requests. Notice that the synchronization between Au and C_i takes place only if C_i is able to provide some information that allows Au to "recognize" C_i as a legal user, which is abstracted with auth_i. Production 2 simply states that a request is sent to the server. For simplicity, we assume that the server does not give back any answer.

Productions for Au simply have to accept authorized clients and give them the server address:

$$x : Mil, u : Mil \vdash Au(x, u) \xrightarrow{(x, \text{auth}_i, \langle u \rangle)} x : Mil, u : Mil \vdash Au(x, u), \quad (3)$$

where i ranges over the indexes of valid clients.

Finally, the server simply accepts requests from clients:

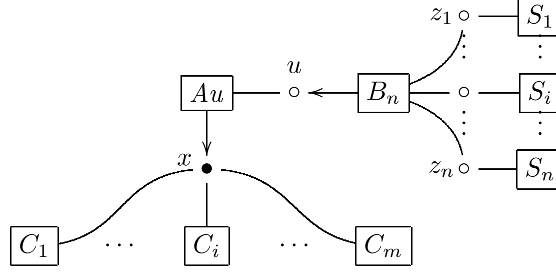
$$u : Mil \vdash S(u) \xrightarrow{(u, \text{req}, \langle\rangle)} u : Mil \vdash S'(u),$$

where S' is used to model the server when it is busy. It becomes again available via the production

$$u : Mil \vdash S'(u) \xrightarrow{(u, \epsilon, \langle\rangle)} u : Mil \vdash S(u),$$

which does not require any synchronization since it corresponds to an internal transition of the server.

Consider that we want to extend our example by making each request to be served by many servers at once. For instance, assume that the request is a search query on the web that clients want to submit to different search engines. This extension can be easily obtained by interposing an edge B_n between Au and the n servers. This system can be represented by the following figure:



B_n will simply acquire the requests from clients and forward them to each server. This behaviour is formally stated by the following production:

$$u : Mil, \boldsymbol{z} : \mathbf{Mil} \vdash B_n(u, \boldsymbol{z}) \xrightarrow{\substack{(u, \text{req}, \langle\rangle),\\ (\boldsymbol{z}, \overline{\text{req}}, \langle\rangle)}} u : Mil, \boldsymbol{z} : \mathbf{Mil} \vdash B_n(u, \boldsymbol{z}).$$

where $\boldsymbol{z} : \mathbf{Mil}$ shortens $z_1 : Mil, \ldots, z_n : Mil$ while with $(\boldsymbol{z}, \overline{\text{req}}, \langle\rangle)$ we denote $(z_1, \overline{\text{req}}, \langle\rangle), \ldots, (z_n, \overline{\text{req}}, \langle\rangle)$.

This solution has a number of advantages:

- the introduction of B_n is completely transparent to the other components, hence we do not have to change the productions for Au, C or S,
- B_n triggers all the servers in parallel.

Nevertheless, there also are some drawbacks:

- if one of the servers crashes or autonomously disconnects from B_n, then B_n is blocked,
- if B_n crashes then all the servers are isolated,
- adding (resp. removing) a server implies to replace B_n with B_{n+1} (resp. B_{n-1}),
- the request is dangling if one of the servers does not accept it.

All these drawbacks are resolved by using weak broadcast synchronizations between the clients and the servers. Instead of introducing edges B_n, we can simply modify productions (1) and (3) as follows:

$$x : Mil \vdash C_i(x) \xrightarrow{(x, \overline{\text{auth}_i}, \langle y\rangle)} x : Mil, y : Bdc \vdash C_i'(y) \tag{4}$$

$$x : Mil, u : Bdc \vdash Au(x, u) \xrightarrow{(x, \text{auth}_i, \langle u\rangle)} x : Mil, u : Bdc \vdash Au(x, u),$$

where Bdc is the weak broadcast SAM on the action {req} (see Example 2). Notice that Au and C_i must accord on the label of the second attachment node of Au. Notice also that now only request actions can be executed on node u.

5 The Mathematics of Heterogeneous SHR

We present now the formal definitions of productions and transitions. In addition to what shown in Section 4, now the label of a transition also contains an idempotent substitution π that allows to merge nodes in the interface by mapping each of them into a standard representative of its equivalence class. Even though not used in this work, we include π since we extend the SHR approach from [5] where it has been exploited.

Definition 4 (SHR Transition). *A* SHR transition *is a relation of the form:*

$$\Gamma \vdash G \xrightarrow{\Lambda,\pi} \Phi \vdash G'$$

where $\Gamma \vdash G$ and $\Phi \vdash G'$ are labelled graphs, $\Lambda : \mathrm{dom}(\Gamma) \to (Act \times \mathcal{N}^*)$ is a total function and $\pi : \mathrm{dom}(\Gamma) \to \mathrm{dom}(\Gamma)$ is an idempotent substitution. If $\Lambda(x) = (a, \boldsymbol{y})$ then $|\boldsymbol{y}| = \mathrm{ar}(a)$. We define $\mathrm{act}_\Lambda(x) = a$, $\mathrm{n}_\Lambda(x) = \boldsymbol{y}$ and

- $\mathrm{n}(\Lambda) = \{z | \exists x. z \in \mathrm{n}_\Lambda(x)\}$ *set of exposed names;*
- $\Gamma_\Lambda = \mathrm{n}(\Lambda) \setminus \mathrm{dom}(\Gamma)$ *set of exposed fresh names.*

We require $\mathrm{dom}(\Phi) = \pi(\mathrm{dom}(\Gamma)) \cup \Gamma_\Lambda$, namely free nodes are never erased ($\supseteq$) and new nodes are bound unless exposed ($\subseteq$). The SAMs associated to new nodes in Φ (nodes in Γ_Λ) can be freely chosen. Instead, for each $x \in \pi(\mathrm{dom}(\Gamma))$, the associated algebra A is $A_1 \diamond \ldots \diamond A_n$ where $\{A_1, \ldots, A_n\}$ are the labels of the nodes in Γ mapped to x by π. Notice that A is well-defined since SAMs form a commutative monoid.

We want to be able to specify productions that can be applied to nodes with different labels, while keeping some control on them. Hence, we introduce types:

Definition 5 (Types). *A* type *t is any non empty set of SAMs.*

Definition 6 (Productions). *A* production *is an SHR transition of the form $x_1 : A_1, \ldots, x_n : A_n \vdash L(x_1, \ldots, x_n) \xrightarrow{\Lambda,\pi} \Phi \vdash G$ where $x_1, \ldots, x_n$ are all distinct and $A_1, \ldots, A_n$ are SAMs.*

A production schema *is a generalized production that has types instead of SAMs as labels for nodes.*

Productions can be derived from production schemas by α-converting the nodes in $\{x_1, \ldots, x_n\} \cup \mathrm{dom}(\Phi)$ and/or by specializing each type t_i into a particular SAM $A_i \in t_i$, provided that for any x_i, $\mathrm{act}_\Lambda(x_i) \in \mathrm{factset}(A_i)$ and that the result is a correct transition (namely, nodes in Φ that are the result of a fusion have the label computed by composing the labels of the merged nodes). We suppose to have for each edge label L of arity n a special idle production schema $\boldsymbol{x} : \boldsymbol{\mathfrak{Alg}} \vdash L(\boldsymbol{x}) \xrightarrow{\Lambda_\epsilon, id} \boldsymbol{x} : \boldsymbol{\mathfrak{Alg}} \vdash L(\boldsymbol{x})$ where $\Lambda_\epsilon(x_i) = (\epsilon, \langle\rangle)$ for each $i \in Int_{|\boldsymbol{x}|}$.

Transitions for SHR are derived by composing productions using the inference rules in the following definition.

Definition 7 (Heterogeneous SHR). *A* heterogeneous SHR rewriting system *consists of a triple* $(\mathcal{A}\mathfrak{lg}, \mathcal{P}, \Gamma \vdash G)$*, where* $\mathcal{A}\mathfrak{lg}$ *is the set of SAMs,* $\mathcal{P}$ *is a set of productions and* $\Gamma \vdash G$ *is the initial labelled graph.*

The set of transitions of $(\mathcal{A}\mathfrak{lg}, \mathcal{P}, \Gamma \vdash G)$ *is the smallest set obtained by applying the inference rules below starting from the productions in* $\mathcal{P}$*. The* computations of $(\mathcal{A}\mathfrak{lg}, \mathcal{P}, \Gamma \vdash G)$ *are sequences of transitions* $\Gamma_i \vdash G_i \xrightarrow{\Lambda_i,\pi_i} \Phi_i \vdash G'_i$*, for* $i \in \omega$*, such that* $\Gamma_0 \vdash G_0 = \Gamma \vdash G$ *and, for each* $i > 0$*,* $\Gamma_i \vdash G_i$ *is* $\Phi_{i-1} \vdash G'_{i-1}$*.*

$$(res)\quad \frac{\Gamma, x : A \vdash G \xrightarrow{\Lambda,\pi} \Phi \vdash G' \quad \mathrm{act}_\Lambda(x) \in \mathrm{factfin}(A) \quad (x\pi = y\pi \wedge x \neq y) \Rightarrow x\pi \neq x}{\Gamma \vdash \nu\, x : A.G \xrightarrow{\Lambda|_{\setminus\{x\}},\pi|_{\setminus\{x\}}} \Phi' \vdash \nu\, Z\; G'}$$

where $Z = \Phi \setminus \Phi'$*.*

$$(new)\quad \frac{\Gamma \vdash G \xrightarrow{\Lambda,\pi} \Phi \vdash G' \quad x \notin \mathrm{dom}(\Gamma) \cup \mathrm{dom}(\Phi) \quad A \in \mathcal{A}\mathfrak{lg}}{\Gamma, x : A \vdash G \xrightarrow{\Lambda \cup \{(x,\epsilon,\langle\rangle)\},\pi} \Phi, x : A \vdash G'}$$

$$(par)\quad \frac{\begin{array}{c}\Gamma_1 \vdash G_1 \xrightarrow{\Lambda_1,\pi_1} \Phi_1 \vdash G'_1 \quad \Gamma_2 \vdash G_2 \xrightarrow{\Lambda_2,\pi_2} \Phi_2 \vdash G'_2 \\ (\mathrm{dom}(\Gamma_1) \cup \mathrm{n}(\Lambda_1)) \cap (\mathrm{dom}(\Gamma_2) \cup \mathrm{n}(\Lambda_2)) = \emptyset\end{array}}{\Gamma_1, \Gamma_2 \vdash G_1|G_2 \xrightarrow{\Lambda_1 \cup \Lambda_2,\pi_1 \cup \pi_2} \Phi_1, \Phi_2 \vdash G'_1|G'_2}$$

$$(merge)\quad \frac{\begin{array}{c}\Gamma, x : A, y : A \vdash G \xrightarrow{\Lambda \cup \{(x,a_1,\boldsymbol{v}_1),(y,a_2,\boldsymbol{v}_2)\},\pi} \Phi \vdash G' \\ (a_1, a_2, (c, \mathrm{Mb})) \in \mathrm{factcmp}(A)\end{array}}{\Gamma, x : A \vdash G\sigma \xrightarrow{\Lambda',\pi'} \Phi' \vdash \nu\, Z\; G'\sigma\rho}$$

where:

- $\sigma = \{x/y\}$;
- $E = \{\boldsymbol{v}_{i_1}[j_1] = \boldsymbol{v}_{i_2}[j_2] \,|\, \mathrm{Mb}([i_1, j_1]) = \mathrm{Mb}([i_2, j_2])\}$;
- $\rho = \mathrm{mgu}(\{(E \cup \mathrm{merge}(\pi))\sigma\})$ *where we choose node names in* $\mathrm{dom}(\Gamma) \cup \{x\}$ *as representatives whenever possible;*
- $\boldsymbol{w}[i] = (\boldsymbol{v}_j[k])\sigma\rho$ *if* $\mathrm{Mb}([j, k]) = i$*,* $i \in Int_{\mathrm{ar}(c)}$*;*
- $\Lambda'(z) = \begin{cases} (c, \boldsymbol{w}) \text{ if } z = x \\ (\mathrm{act}_\Lambda(z), (\mathrm{n}_\Lambda(z))\sigma\rho) \text{ for each } z \in \mathrm{dom}(\Gamma) \end{cases}$
- $\pi' = \rho|_{\mathrm{dom}(\Gamma) \cup \{x\}}$;
- $\mathrm{dom}(Z) = \mathrm{dom}(\Phi)\sigma\rho \setminus \mathrm{dom}(\Phi')$;
- *the label of each node* $x \in \mathrm{dom}(Z) \cup \mathrm{dom}(\Phi')$ *is computed as follows:* x *is the representative according to* ρ *of an equivalence class* $\{x_1, \ldots, x_n\}$ *of nodes which have in* Φ *labels* $A_1, \ldots, A_n$*. Then the label of* x *is* $A_1 \diamond \ldots \diamond A_n$*.*

Rule (res) binds nodes, allowing only complete actions (w.r.t. their SAMs) to take place on them. According to the last condition of (res), the bound node must not be the representative of the equivalence class induced by π when the class is not trivial. Nodes extruded just on the bound node must be bound after the transition, and thus they are in Z (the labels are preserved).

Rule (new) allows to add an isolated node x to the interface; initially, on x only the trivial action ϵ can be done.

Rule (par) performs the union of two transitions provided that they have disjoint sets of free names (accounting also for newly generated names).

Rule (merge) is the rule for synchronization. It allows to compute the effect of merging two nodes x and y with synchronizations $(a_1, \boldsymbol{v}_1)$ and $(a_2, \boldsymbol{v}_2)$ respectively on them, provided that they are labelled with the same SAM A, which will label also the resulting node. The synchronization is allowed iff there exists a triple $(a_1, a_2, (c, \mathrm{Mb})) \in \mathrm{factcmp}(A)$. In this case set E is computed, which accounts for merging names that are mapped to the same position by Mb (note that merges are performed even if the resulting representative is not attached to action c). We then compute ρ by means of an mgu on $E \cup \mathrm{merge}(\pi)$ after having fused nodes according to σ. Notice that $\mathrm{merge}(\pi)$ accounts for the node fusions due to π and that ρ also chooses a representative for each equivalence class. If at least one of the members of the class is in $\mathrm{dom}(\Gamma) \cup \{x\}$, then one of them must be chosen (otherwise undesired renamings of nodes may happen). After that, the new vector $\boldsymbol{w}$ is generated by choosing for each position the representative of the corresponding equivalence class. We can then compute the new synchronization function Λ', which takes into account the performed merges. Merges on nodes in the interface are traced by π'. Nodes that are no longer extruded (because the synchronization discards them) are bound. The SAMs associated to nodes are preserved from the premise for nodes that are not merged, and are computed using the operator of composition of SAMs otherwise. The result does not depend on the structure of the derivation, as witnessed by the following proposition:

Proposition 1. *Given a transition* $\Gamma \vdash G \xrightarrow{\Lambda,\pi} \Phi \vdash G'$*, the labels in* Φ *of nodes in* $\mathrm{dom}(\Phi) \cap \mathrm{dom}(\Gamma)$ *depend only on* Γ *and* π*.*

Proof. By induction on the structure of the derivation.

Example 3 (Dynamically computing SAMs). Consider the example of § 4. Using weak broadcast communications between clients and servers is "statically" determined by the productions of A and C_i which also couple the behaviour of these components. Indeed, if $Mil \diamond Bdc = Bdc$, then rule (merge) ensures that the synchronization of productions (4) and (3) yields the expected result, namely, the attachment node of servers uses a weak broadcast SAM, since C_i requests it to do so.

6 A Further Example

We consider a scenario where multiple processes/hosts are connected through links with different features. Mainly, we consider two characteristics: the maximum packet size, which can be either 4kb or 16kb, and the presence/lack of some error-detecting mechanism e.g., the CRC algorithm. Before starting the real communication, two processes create a logical communication channel between them.

This channel supports 16kb packets and/or error-detecting capabilities only if all the underlying channels do.

We model this scenario using eight SAMs obtained by mixing three distinct features, namely:

- packet sizes (4kb/16kb),
- error-detecting mechanism (yes/no),
- control or communication link.

The first two items correspond to the aforementioned characteristics of links, while the last one specifies whether the link is used for exchanging control messages or for data. For simplicity we suppose that control messages are short and thus can be encoded using some error-correcting code. Hence, control messages are always correctly delivered. Since all the combinations of the three features are possible, we conventionally name the SAMs by means of expressions like CTR_4 or $\mathrm{COM}_{16}^{\surd}$ that respectively denote a control link without error-detecting mechanism and with 4kb maximum packet size and a communication link with error-detecting mechanism and 16kb maximum packet size.

The control SAMs provide essentially normal Milner communication (where we drop the distinction between action and coaction) for control messages. The SAMs for communication without error-detecting mechanism provide faulty Milner synchronization, i.e., the result of a synchronization can be either τ or err. In the former case, name passing is performed while no name passing is performed in the latter. Note that, in these SAMs, the processes cannot detect the result of synchronizations (unless they perform additional interactions). Communication actions can be of two kinds, in_4 for packets of 4kb and in_{16} for packets of 16kb. Corresponding output actions are provided. The same τ and err actions are used in both the cases. Both the kinds of synchronization are allowed by the SAM 16kb, while just the 4kb-size packets are allowed for SAMs 4kb. Finally, error-detecting SAMs allow two different kinds of input, $in_s^{\surd}$ which synchronizes giving τ and in_s which synchronizes giving a detected error, where $s \in \{4\mathrm{kb}, 16\mathrm{kb}\}$.

To clarify all that we will now formally define the SAM $\mathrm{COM}_{16}^{\surd}$:

- $N = \mathrm{COM}_{16}^{\surd}$;
- $Act = \{in_4^{\surd}, in_4, out_4, in_{16}^{\surd}, in_{16}, out_{16}, \tau, err, \epsilon\}$;
- for simplicity we suppose that all actions but τ, err and ϵ have arity 1 while these three have arity 0;
- $Fin = \{\tau, err, \epsilon\}$;
- $(in_4^{\surd}, out_4, (\tau, MP_{1,1})) \in ActCmp$,
  $(in_4, out_4, (err, MP_{0,0})) \in ActCmp$,
  $(in_{16}^{\surd}, out_{16}, (\tau, MP_{1,1})) \in ActCmp$,
  $(in_{16}, out_{16}, (err, MP_{0,0})) \in ActCmp$,
  $(\epsilon, \epsilon, (\epsilon, MP_{0,0})) \in ActCmp$.

We can now define the function of algebra composition. We will define a partial order on our SAMs, and the greatest lower bound will be our composition

operator. The order is defined component-wise on the three features of SAMs over which the following orders are considered: 16kb > 4kb, the presence of an error-detecting mechanism is greater than its absence and $CTR > COM$. Note that the set of SAMs with the resulting operation forms a commutative monoid.

We consider as types all the singletons, the universal type $\mathcal{A}\mathfrak{lg}$ and the type CT that contains the four control SAMs.

For simplicity we consider that our system contains two kinds of subsystems: end systems $x \vdash \mathit{Idle}(x)$ and routers $x, y, z \vdash R(x, y, z)$. End systems can start a communication with the production schema:

$$x : CT \vdash \mathit{Idle}(x) \xrightarrow{(x, connect, \langle y \rangle)} x : CT, y : \mathrm{COM}_{16}^{\checkmark} \vdash \mathit{Active}(x, y) \qquad (5)$$

After connecting to another system, an *Idle* process becomes *Active*, i.e. able to send/receive data over the links built toward other systems. For lack of space, we do not formally model their behaviour.

We now show the productions for routers. We write just a meta-rule, with type meta-variables t_1 and t_2 to be instantiated with each singleton containing a control SAM. Furthermore the production must be written for each permutation of attachment nodes.

$$\begin{array}{c} x : t_1, y : t_2, z : CT \vdash R(x, y, z) \xrightarrow{(x, connect, \langle w \rangle), (y, connect, \langle w \rangle)} \\ x : t_1, y : t_2, z : CT, w : (\mathrm{COM}_{16}^{\checkmark} \diamond t_1 \diamond t_2) \vdash R(x, y, z) \end{array} \qquad (6)$$

We can now show some examples. Let us consider as starting graph:

$$\vdash \begin{array}{c} (\nu\, x_1 : \mathrm{CTR}_{16}^{\checkmark}, x_2, x_3 : \mathrm{CTR}_{16}, x_4 : \mathrm{CTR}_4, x_5, x_6 : \mathrm{CTR}_4^{\checkmark}). \\ (\mathit{Idle}(x_1) | R(x_1, x_2, x_4) | R(x_2, x_3, x_5) | \mathit{Idle}(x_3) | R(x_4, x_5, x_6) | \mathit{Idle}(x_6)), \end{array}$$

where $x_1, \ldots, x_n : A$ shortens $x_1 : A, \ldots, x_n : A$.

For instance, we can build a communication channel between $\mathit{Idle}(x_1)$ and $\mathit{Idle}(x_3)$ using productions (5) and (6). The algebra for the resulting connection is COM_{16}, obtained via $\mathrm{COM}_{16}^{\checkmark} \diamond \mathrm{COM}_{16} \diamond \mathrm{COM}_{16}$. In that case we obtain the graph:

$$\vdash \begin{array}{c} (\nu\, x_1 : \mathrm{CTR}_{16}^{\checkmark}, x_2, x_3 : \mathrm{CTR}_{16}, x_4 : \mathrm{CTR}_4, x_5, x_6 : \mathrm{CTR}_4^{\checkmark}, y : \mathrm{COM}_{16}) \\ (\mathit{Active}(x_1, y) | R(x_1, x_2, x_4) | R(x_2, x_3, x_5) | \mathit{Active}(x_3, y) | R(x_4, x_5, x_6) | \mathit{Idle}(x_6)) \end{array}$$

Similarly we can build a connection between $\mathit{Idle}(x_1)$ and $\mathit{Idle}(x_6)$, but this time the resulting channel will use the algebra COM_4. When either of these connections has been established, communication can happen using that channel. Notice that the label of the channel is computed from the constraints imposed by interacting routers.

7 Conclusions

We have presented a framework for modelling heterogeneous distributed systems using SHR. In addition to standard SHR properties, namely the ability to build

a compositional description of system behaviour, our extension allows to deal with systems where different kinds of communication/synchronization policies coexist, as usually happens in WANs and overlay networks. From the technical point of view the main points are the labelling of nodes with SAMs and the management of these labels when nodes are communicated and merged, which is obtained by requiring a commutative monoid structure on the set of SAMs.

As future work we plan to analyze the behavioural properties of SHR systems by defining a suitable bisimulation which is a congruence w.r.t. composition of graphs. We also want to develop a framework that merges the advantages of SHR and bigraphs, thus allowing a compositional description of systems with both a communication and a location structure. Another important point is to build an implementation of SHR systems, both for modelling system evolution (and also performing behavioural analysis using model-checking) and for programming real systems. In this second case SHR must be integrated with a standard programming language such as Java or C++, thus enabling to write normal code for computation and using some flavour of SHR (probably in a more process-calculi like style) for coordination.

References

1. I. Castellani and U. Montanari. Graph Grammars for Distributed Systems. In H. Ehrig, M. Nagl, and G. Rozenberg, editors, *Proc. 2nd Int. Workshop on Graph-Grammars and Their Application to Computer Science*, volume 153 of *LNCS*, pages 20–38. Springer, 1983.
2. R. De Nicola, G. Ferrari, U. Montanari, R. Pugliese, and E. Tuosto. A Formal Basis for Reasoning on Programmable QoS. In N. Dershowitz, editor, *International Symposium on Verification – Theory and Practice – Honoring Zohar Manna's 64th Birthday*, volume 2772 of *LNCS*, pages 436 – 479. Springer, 2003.
3. P. Degano and U. Montanari. A model of distributed systems based on graph rewriting. *JACM*, 34:411–449, 1987.
4. H. Ehrig, M. Pfender, and H. J. Schneider. Graph grammars: an algebraic approach. In *Proceedings IEEE Conference on Automata and Switching Theory*, pages 167–180, 1973.
5. G. Ferrari, U. Montanari, and E. Tuosto. A LTS Semantics of Ambients via Graph Synchronization with Mobility. In *ICTCS*, volume 2202 of *LNCS*. Springer, 2001.
6. G. Ferrari, U. Montanari, and E. Tuosto. Graph-based Models of Internetworking Systems. In T. Aichernig, Bernhard K. Maibaum, editor, *Formal Methods at the Crossroads: from Panaces to Foundational Support*, volume 2757 of *LNCS*, pages 242 – 266. Springer, 2003.
7. D. Hirsch. *Graph Transformation Models for Software Architecture Styles*. PhD thesis, Departamento de Computación, UBA, 2003. http://www.di.unipi.it/~dhirsch.
8. D. Hirsch, P. Inverardi, and U. Montanari. Reconfiguration of Software Architecture Styles with Name Mobility. In A. Porto and G.-C. Roman, editors, *Coordination 2000*, volume 1906 of *LNCS*, pages 148–163. Springer, 2000.
9. D. Hirsch and U. Montanari. Synchronized hyperedge replacement with name mobility: A graphical calculus for name mobility. In *CONCUR*, volume 2154 of *LNCS*, pages 121–136, Aalborg, Denmark, 2001. Springer.

10. O. Jensen and R. Milner. Bigraphs and transitions. *SIGPLAN Not.*, 38(1):38–49, 2003.
11. I. Lanese and U. Montanari. Software architectures, global computing and graph transformation via logic programming. In L. Ribeiro, editor, *Proc SBES'2002 - 16th Brazilian Symposium on Software Engineering*, pages 11–35. Anais, 2002.
12. I. Lanese and U. Montanari. Synchronization algebras with mobility for graph transformations. In *Proc. FGUC'04 – Foundations of Global Ubiquitous Computing*, ENTCS, 2004. To appear.
13. R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, I and II. *Inf. and Comp.*, 100(1):1–40,41–77, September 1992.
14. U. Montanari and F. Rossi. Graph rewriting and constraint solving for modelling distributed systems with synchronization. In P. Ciancarini and C. Hankin, editors, *Proceedings of the First International Conference COORDINATION '96, Cesena, Italy*, volume 1061 of *LNCS*. Springer, April 1996.
15. D. Sangiorgi and D. Walker. *The π-Calculus: a Theory of Mobile Processes*. Cambridge University Press, 2002.
16. E. Tuosto. *Non-Functional Aspects of Wide Area Network Programming*. PhD thesis, Dipartimento di Informatica, Università di Pisa, May 2003. TD-8/03.
17. G. Winskel. Synchronization trees. *TCS*, 34:33–82, May 1985.