The Synchronised Hyperedge Environment

Ivano Talamo
CASPUR
Consorzio interuniversitario per le Applicazioni di Supercalcolo
Per Universitá e Ricerca
italamo@caspur.it

Alessandro Tiberi
Dipartimento di Informatica, "La Sapienza"
Via Salaria, 113, Roma
tiberi@di.uniroma1.it

Pietro Cenciarelli
Dipartimento di Informatica, "La Sapienza"
Via Salaria, 113, Roma
cenciarelli@di.uniroma1.it

Abstract

(versione 4) We introduce the Synchronised Hyperedge Environment, SHE, a tool for developing, analysing and automatically verifying distributed and concurrent systems. SHE supports a visual, declarative style of programming based on a graph rewrite system called Synchronising Graphs (SG) [3, 4], a general semantical framework which has been used for interpreting various process calculi, such as Mobile Ambients, the distributed CCS and Fusion. After describing the system's architecture, we develop two applications: The first offers a simple declarative solution to the problem of syntactic unification, and shows how flexible the system is in supporting both textual and graphical representation of data. The second example is a classical problem in distributed programming: the leader election. The proposed solution is proven correct both by a mathematical proof and by automatic verification through model checking.

1 Introduction

We introduce the *Synchronised Hypergraph Environment* (*She*),[1] a tool for assisting distributed software development and verification. The system provides an interactive environment based on graph rewriting, where applications are developed declaratively and computation can be visualised as a sequence of graph transformations. The user can explore the state space of the application under construction, and search for states satisfying safety conditions, as well as non-local properties, such as having a specified number of next states. Execution paths can be selected and the corresponding run can be animated. It is possible, for example, to visualise the movement of the forks in between dining philosophers, or the routing of a message in a network of communicating agents.

She is based on *Synchronising Graphs* (SG) [3], a powerful system of graph rewriting inspired by [7]. SG have been used by the authors to model various process calculi, such as Mobile Ambients, the distributed CCS and Fusion (ibid.). The simplicity of the encoding and its mathematical tractability (e.g. in proving operational correspondence) suggests that SG is well suited as common semantic framework for calculi of mobility: by implementing specific front-ends, and using SG as intermediate language, She can work as a multilingual interpreter and verifier, a novelty with respect to similar systems ([6, 10] to cite a few). In the present paper we develop two applications. The first offers a simple, declarative solution to the problem of *syntactic unification* [11], where computation is distributed over the nodes of a *term graph*. The example shows that SG specifications are compact (28 characters alltogether [**?**]) and that She can easily be tailored to suit the specific needs of different applications. The second example, the *leader election* problem, is a classic in distributed programming. The proposed implementation is proven correct both by a mathematical proof (showing the tractability of synchronising graphs) and by automatic verification through model checking, showing an integration of the latter (which is more often used for verifying fully developed systems) within the framework of (distributed) program development.

[1] available at HTTP://BRIANTB.UNIXCAB.ORG/SHETEMP/

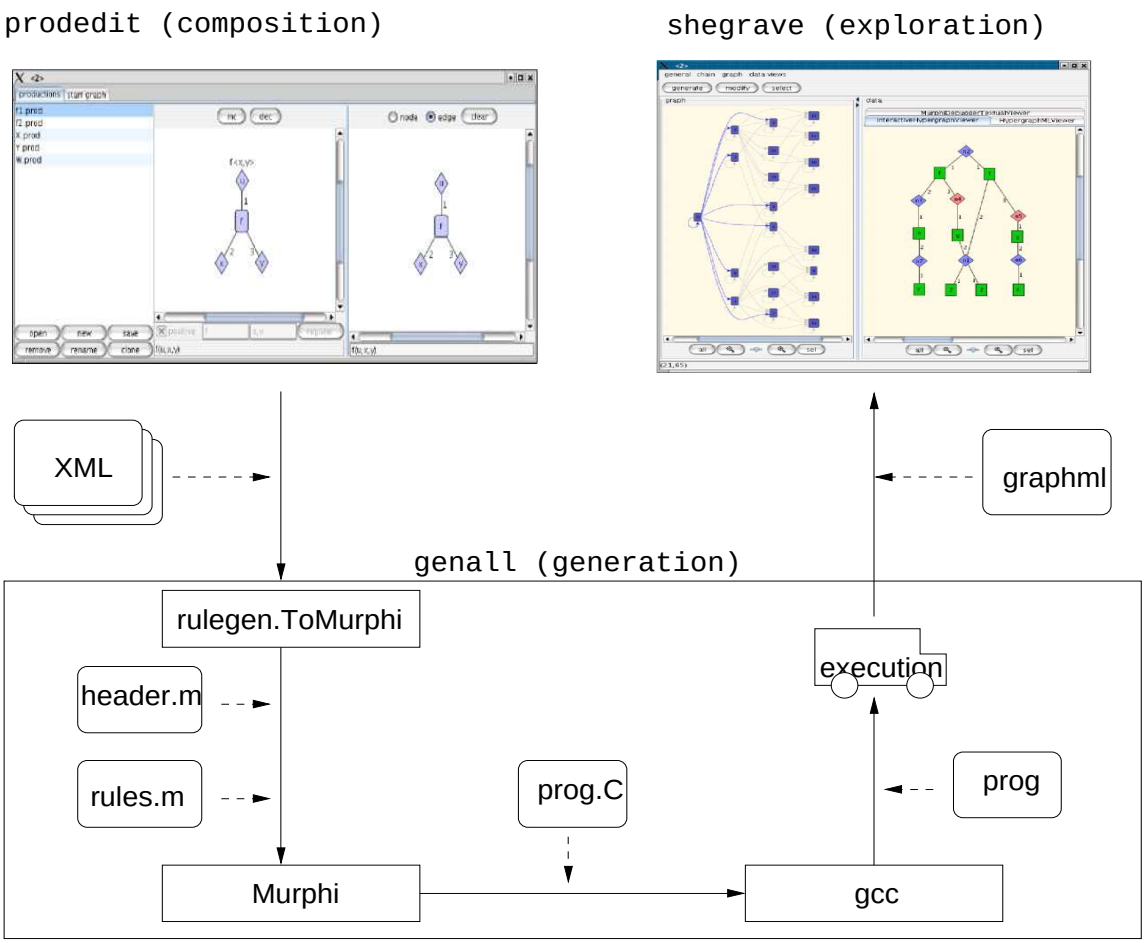

Figure 1. The architecture of She

2 The Synchronised Hyperedge Environment

The Synchronised Hyperedge Environment (figure 1) is composed of three modules corresponding to three different stages in the development of a SG application:

- specification of a system,
- generation of its transitions,
- verification and analysis of computation.

The first component, `prodedit`, is a tool that allows the user to specify a set of axioms and an initial graph. Its output is a set of XML files that are parsed by the next component, `genall`, which generate the graph $\mathcal{S}$ of all reachable states. The nodes of $\mathcal{S}$ represent synchronising graphs which can be obtained from the initial graph by means of rewriting, while edges represent transitions derivable from the axioms (see section 3). The module `genall` uses the Murphi model checker [8] to generate a *graphml* representation of the state graph of the system. Graphml [2] is a standard and widely used XML format for describing graphs. The module `shegrave` allows system verification by analysis of the state graph.

The fact that the different components communicate exclusively by exchanging files makes SHE an open application, where users can add their own modules. Moreover, the choice of XML facilitates the development of add-on modules as it is largely supported format.

Specification of a system. A SG application consists of a set of axioms and an initial graph. The interactive module `prodedit` allows users to specify both in a visual fashion. An axiom is created by drawing (point 'n click, drag 'n drop) left and right hand side of the axiom in distinct panels, and by adding actions over nodes. Once a production

has been drawn the tool checks that it is coherent. Productions can be deleted, added and duplicated. Initial graphs are created in much the same way: the user creates and links nodes and hyperedges. The tool produces an XML file for each axiom and one for the initial graph.

Generation of the transitions. Once a SG application has been specified it is possible to run it, thus generating its state graph, which includes all possible computations originating from the initial graph. The state graph is generated by the Murphi model checker. Murphi allows the user to describe a system by a specification consisting of an initial state and a set of conditional rule to perform state change. From the XML file produced during the previous phase, SHE creates a Murphi specification where the starting state is the initial graph and every production is translated into a Murphi rule. Next the specification is compiled to obtain an executable which actually explores the state space. The output of the exploration is the state graph, in graphml format.

System analysis. The state graph can be analyzed and properties can be verified by the third software component, `shegrave`. For the development of this component, we wanted a tool where users could visually explore the state graph and select states both by using the mouse or by submitting queries (see below). We chose to develop a flexible tool, the *Graph Viewer and Explorer* (GraVE), with the general capability of exploring a graph with information attached to nodes, and then to tailor it to the specific domain of SHE, thus producing `shegrave`. In particular, Grave includes:

- a component for the generation of a graph G;
- a component for the visualization of G and
- a component for the visualization of the information associated with the nodes of G.

All these components are configurable, that is the user can create its own Java classes to generate the graph, visualise it, and view the information attached to its nodes. The component for the generation of the graph is not a single Java class but a pipeline of classes, each one taking input from the previous, performing some actions on it, and then passing it forward. The pipeline components can be reused in other applications, this speeding up development process. For example, a pipeline used in `shegrave` would typically be as follows:

- the first element prompts the user for a graphml file representing a SG state graph, and generates a Java object representing the given graph;
- the second element applies a layout to the graph;
- the third element applies label i to the i-th node;
- the fourth element resizes nodes according to their labels.

Further the pipeline classes can insert items in the menu of the application, thus expanding its functionalities. For example, GraVE provides a class that acts as a pass-through element in the pipeline, but adds a "Save" item to the menu, to save the passed-through graph to a file in the graphml format. Some pipeline elements provide both the functionality. For example the layouter applies a default layout to the graph but also adds an item to the application menu that allows the user to select a different layout to the graph.

GraVE already provides some of these general components, like classes to open a graph from a file, to layout a graph, to save it on a file, and various classes to change graphical features of the graph.

GraVE also offers its users a tool for selecting nodes by means of queries. These are expressions involving boolean operators (AND, OR, NOT) as well as graph-specific operators, such as for select nodes with zero incoming/outcoming edges, and so forth. The set of available operators can be expanded by a plugin method: if a user wants a new operator he/she simply writes a Java class that implements the plugin interface. Each operator selects some nodes on the state graph and this selection can depend on the selection of other operators. For example, the *RAND* operator selects nodes randomly, *SOURCE* select all the nodes that have only outcoming edges (sorce nodes), and *AND(RAND,SOURCE)* selects randomly a source node.

These expression can be expressed both in a graphical and in a textual way, and can be saved on files, for later use of exchange with other users.

3 Synchronising Graphs

This short presentation of SG is in great part borrowed from [4, 5], to which we refer the reader for more details and examples.

Synchronising Graphs were proposed in [3] as a model of process interaction in a network environment. The model is based on *hyperedge replacement* [7, 9], a form of graph transformation where edges, representing processes, interact by synchronising action and co-action pairs at specific synchronisation points, the nodes, representing communication channels. The behaviour of parallel, possibly distributed systems is specified in SG by a set of axioms. System transitions are derived by means of inference rules implementing agent interaction (synchronisation) and resource encapsulation (restriction).

Graphs. Let $\mathcal{N}$ be a set of *nodes*, which we consider fixed throughout. A *graph* $G = (E, G, R)$ consists of a set E

of *hyperedges* (or just *edges*), an attachment function $G : E \to \mathcal{N}^*$ and a set $R \subseteq |G|$ of nodes, called *restricted*, where

$$|G| = \{x \in \mathcal{N} \mid \exists\, e \in E \text{ s.t. } G\,e = x_1 \dots x_n \text{ and } x = x_i\}$$

is the set of nodes of the graph. We denote by $\mathit{res}\,(G)$ the set of restricted nodes of G, and by $\mathit{fn}\,(G)$ the set $|G| - \mathit{res}\,(G)$ of *free* nodes of G. We write $e(\vec{x})$ for an edge of a graph G such that $G\,e = \vec{x}$. Moreover, we let $\nu x\, G$ denote the graph $(E, G, R \cup \{x\})$ when $x \in |G|$, while $\nu x\, G = G$ otherwise. If (E, G, R) and (D, F, S) are graphs such that $E \cap D = |G| \cap S = |F| \cap R = \emptyset$, we write $G|F$ the graph $(E \cup D, G + F, R \cup S)$, whose attachment function $G + F$ maps $e \in E$ to $G\,e$ and $d \in D$ to $F\,d$.

Transitions. Let $\mathit{Act} = \{a, b, \dots\} \cup \{\overline{a}, \overline{b}, \dots\}$ be a set of *actions* and *co-actions* (overlined), and let $\overline{\overline{a}}$ denote a. We write Act^+ the set $\mathit{Act} \times \mathcal{N}^*$. Given $(a, \vec{x})$ in Act^+, we call the components of $\vec{x}$ *arguments* of a. A *pre-transition* Λ of a graph G to a graph H, written:

$$G \xrightarrow{\Lambda} H,$$

is a relation $\Lambda \subseteq \mathcal{N} \times \mathit{Act}^+$ such that $\mathit{dom}\,(\Lambda) \subseteq |G|$. We denote with $|\Lambda|$ the set of arguments appearing in Λ. We write $(x, a, \vec{y})$ for an element $(x, (a, \vec{y}))$ of Λ, and (x, a) when $\vec{y}$ is the empty sequence. Intuitively, $(a, \vec{y}) \in \Lambda\, x$ expresses the occurrence of action a at node x. In SG the occurrence of both $(a, \vec{y})$ and $(\overline{a}, \vec{z})$ at x triggers a synchronisation between two agents (edges) of the graph, what is traditionally represented by a *silent* action τ. When such is the case the synchronising agents may exchange information. This is implemented in SG by unifying the lists $\vec{y}$ and $\vec{z}$ of parameters, which are required to be of the same length. Only two agents at a time may synchronise at one node. Moreover, if an action occurs at a restricted node, then it *must* synchronise with a corresponding co-action, as we consider *observable* the unsynchronised actions. A restricted node may be "opened" by unifying it with an argument of an observable action, or with a node which is not restricted.

The above requirements are formalised as follows. An *action set* is a relation $\Lambda \subseteq \mathcal{N} \times \mathit{Act}^+$ such that, for all nodes x, $\Lambda\, x$ has *at most* two elements and, when so, it is of the form $\{(a, \vec{y}), (\overline{a}, \vec{z})\}$, where the lengths of vectors $\vec{y}$ and $\vec{z}$ coincide. Given an action set Λ, we denote by $\overset{\Lambda}{=}$ the smallest equivalence relation on nodes such that, if $(x, a, y_1 y_2 \dots y_n)$ and $(x, \overline{a}, z_1 z_2 \dots z_n)$ are in Λ, then $y_i \overset{\Lambda}{=} z_i$, for $i = 1 \dots n$. The *dangling* nodes of an action set Λ are arguments of unsynchronised actions. More precisely they are elements of the set $\{x \mid \exists\, y \text{ s.t. } \Lambda\, y = \{(a, z_1 \dots z_n)\} \text{ and } x \overset{\Lambda}{=} z_i\}$. A predicate $\mathit{opens}\,(\Lambda, x, G)$ is defined to hold precisely when either x is dangling in Λ or $[x]_{\overset{\Lambda}{=}} \not\subseteq \mathit{res}\,(G)$. A *transition* is a pre-transition $G \xrightarrow{\Lambda} H$ such that:

1. Λ is an action set;
2. if a node x is restricted in G then $|\Lambda\, x| \neq 1$;
3. if a node x occurs in $H \cap (|G| \cup |\Lambda|)$, then $x \in \mathit{fn}\,(H)$ if and only if $\mathit{opens}\,(\Lambda, x, G)$.

An *identity* is a transition of the form $G \xrightarrow{\emptyset} G$. We say that an action a is *observed* at node x during a transition Λ if $\Lambda\, x$ is a singleton $\{(a, \vec{y})\}$. The first clause above expresses the coherence of synchronisation; the second says that no actions can be observed at restricted nodes; the third states the conditions under which a node x, possibly restricted in G, can occur free in H.

Inference rules. Let $f : \mathcal{N} \to \mathcal{N}$ be a function on nodes and let (E, G, R) be a graph. We write fG the graph (E, fG, fR) obtained by substituting all nodes x in G with fx, that is, for all $e \in E$, if $G\,e = x_1 \dots x_n$ then $(fG)\,e = fx_1 \dots fx_n$. A function $f : \mathcal{N} \to \mathcal{N}$ is said to *agree* with an equivalence relation φ on $\mathcal{N}$ if, as a set of pairs, it is a subset of φ, that is if $(x, fx) \in \varphi$, for all nodes $x \in \mathcal{N}$. A *unifier* of φ is a function ρ which agrees with φ and such that $|\rho\,[x]| = 1$ for all x. By a slight abuse, we say that a function agrees with (or unifies) Λ to mean that it agrees with (unifies) $\overset{\Lambda}{=}$.

In SG, synchronisation is subject to a non-interference condition: two transitions $G \xrightarrow{\Lambda} H$ and $F \xrightarrow{\Theta} K$ can be synchronised provided they are disjoint and that nodes appearing as arguments in Λ (Θ) are not $\mathit{res}\,(F)$ ($\mathit{res}\,(G)$). Formally two transitions $G \xrightarrow{\Lambda} H$ and $F \xrightarrow{\Theta} K$ are said to be *non-interfering*, written $\Lambda \# \Theta$, when:

- $\Lambda \cap \Theta = \emptyset$, and moreover
- $|\Lambda| \cap \mathit{res}\,(F) = |\Theta| \cap \mathit{res}\,(G) = \emptyset$.

The rules of the system of synchronising graphs are:

$$[\text{ sync }] \quad \frac{G \xrightarrow{\Lambda} H \qquad F \xrightarrow{\Theta} K}{G \mid F \xrightarrow{\Lambda \cup \Theta} \rho(H|K)} \quad \begin{array}{l}(\text{if } \Lambda \# \Theta \text{ and}\\ \rho \text{ unifies } \Lambda \cup \Theta)\end{array}$$

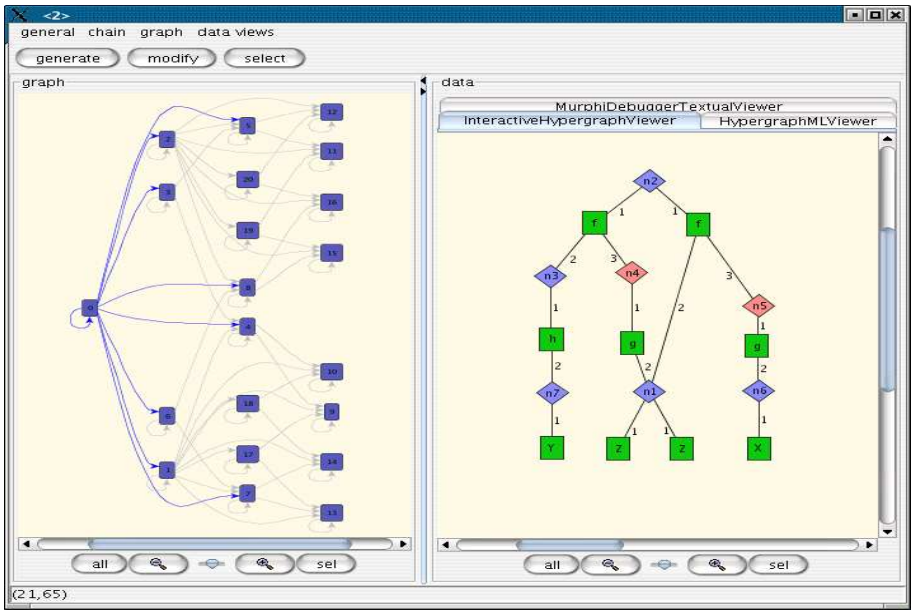

Figure 2. the root.

$$[\text{ open }]\quad \frac{G \xrightarrow{\Lambda} H}{\nu x\, G \xrightarrow{\Lambda} H} \quad (\text{if } \mathit{opens}\,(\Lambda, x, \nu x\, G))$$

$$[\text{ res }]\quad \frac{G \xrightarrow{\Lambda} H}{\nu x\, G \xrightarrow{\Lambda} \nu\rho(x)\, \rho(H)} \quad (\text{if } \neg\mathit{opens}\,(\Lambda, x, \nu x\, G) \text{ and } \rho \text{ unifies } \Lambda)$$

Note that the applicability of [sync] is implicitly constrained by $\Lambda \cup \Theta$ being a transition (e.g. Λ and Θ cannot issue different actions at a same node). Similar considerations apply to [open] and [res].

An *axiom* is a transition $G \xrightarrow{\Lambda} H$ such that $H = \rho H$ for some unifier ρ of Λ. This condition, stating that all nodes unified by Λ are fused in H, is preserved by the inference rules, and it is therefore satisfied by all transitions derived from axioms. Given a set $\mathcal{T}$ of axioms, a $\mathcal{T}$*-computation*, or just *computation* for short, is a sequence of transitions $G_0 \xrightarrow{\Lambda_1} G_1 \xrightarrow{\Lambda_2} \dots$ each of which is derived from the axioms in $\mathcal{T}$.

Actually SHE employs a slightly limited version of SG in which there is no node restriction and all actions have to be synchronised. Also, instead of axioms, productions are used. These are a special kind of transition whose left hand side is a single edge with every tentacle attached to a different node. Productions can be instantiated as long as the result of the instantiation process remains a transition.

4 Syntactic unification

Here we develop a SG theory of *syntactic unification* [11] in SHE. A full length description of this application is given in [4], from which some material used in this section derives, and to which we refer for a more detailed mathematical treatment.

A syntactic unifier of two first order terms t and t' is a substitution of terms for variables making t and t' identical. In our running example we shall consider $t = f(h(y), g(z))$ and $t' = f(z, g(x))$; the function ρ mapping every variable to itself except for $\rho(x) = \rho(z) = h(y)$ is a unifier of t and t', while the result of unification is $f(h(y), g(h(y)))$. The substitution ρ is in fact a *most general* unifier of the two terms, and we shall assume this notion as understood. Let $\mathcal{F} = \{f, g, h \dots\}$ and $\mathcal{V} = \{x, y, z \dots\}$ be disjoint sets of *symbols*, called respectively *functions* and *variables*. We represent first order terms by means of *S-graphs*, that is synchronising graphs whose nodes are variables and whose edges are labelled by symbols. We use a bold $\mathbf{s}$ to denote an edge labeled by a symbol s. A family of functions $[\![\ \_\ ]\!]_x$ indexed by nodes translates terms into *S*-graphs:

$$[\![x]\!]_x = \mathbf{x}(x)$$
$$[\![f(t_1, \dots t_n)]\!]_x = \mathbf{f}(x\, y_1 \dots y_n) \mid [\![t_1]\!]_{y_1} \mid \dots \mid [\![t_n]\!]_{y_n},$$

where $y_i = t_i$ when t_i is a variable, or otherwise y_i is a new node. The node to which the first tentacle of an edge is

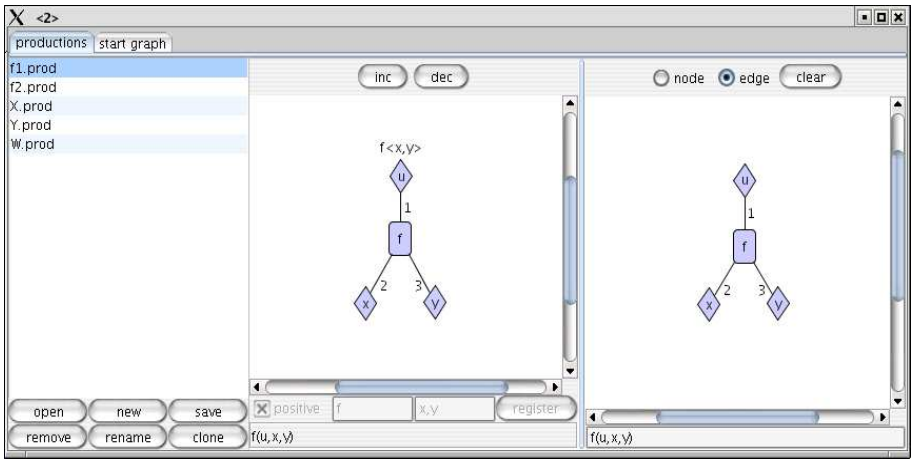

Figure 3. an axiom

attached is called the *result node* of the edge. We define the S-graph $G_{t=t'}$ corresponding to an equation $t = t'$ between terms of T to be either $\mathbf{x}(x) \,|\, \mathbf{y}(x)$, when both $t = x$ and $t' = y$ are variables, or else $[\![t]\!]_x \mid [\![t']\!]_x$, where x is new if neither t nor t' are variables. The right panel in figure 2 depicts $G_{t=t'}$ for $t = f(h(y), g(z))$ and $t' = f(z, g(x))$. (Using the internal naming discipline of SHE, nodes are called $n1 \ldots n7$. This is of course uninfluential.)

The problem of unifying two terms t and t' is solved by a computation of the graph $G_{t=t'}$ within the theory of S-graphs. The theory features a set $\mathit{Act} = \{f, \overline{f}, \ldots x, \overline{x} \ldots\}$ of actions, including all symbols and their complements (co-actions). The axioms of the theory include the identities and all instances of the following axiom schemes.

$$\mathbf{s}(x\,\vec{y}) \xrightarrow{x,s,\vec{y}} \mathbf{s}(x\,\vec{y})$$
$$\mathbf{s}(x\,\vec{y}) \xrightarrow{x,\overline{s},\vec{y}} \emptyset$$

Figure 3 shows the axiom $\mathbf{f}(u\,x\,y) \xrightarrow{u,f,x\,y} \mathbf{f}(u\,x\,y)$, an instance of the first scheme above, as visualised by `prodedit`, the module in SHE which supports interactive generation of axioms. The module features three panels which, proceeding from left to right we shall call A, B and C. The XML file representing the axiom, `f1.prod`, is selected from A, where other such files available for editing are listed. The panels B and C visualise respectively the left and the right hand sides of the axiom, and feature, at the bottom, a textual representation of the corresponding graph (which coincide in this case). The actions, together with their arguments, are written in B beside the associated node. Hence, above node u we find the action $\mathbf{f}$ and the pair $\langle x, y\rangle$ of arguments. These data are edited from the text fields just below the horizontal scroll bar of panel B. The `positive` check box specifies whether the given symbol is to be taken as action or co-action.

As explained in section 2, once the axioms of the theory and an initial graph are generated, the corresponding XML files are compiled into a Murphi specification and run through the model checker. Murphi's log file is then interpreted by the module `shegrave` to produce the graph of all reachable states. The nodes of this graph represent states of the computation, that is synchronising graphs obtained by the rewriting process, while the edges represent transitions. Figure 2 shows a screenshot of `shegrave`'s graphical interface.

The panel to the left, called the *graph* panel, depicts the state graph of our running example. By clicking on a specific node, the corresponding synchronising graph is visualised in the panel to the right, the *data* panel, while all possible next states are highlighted. In the picture, the "root" of the computation (the leftmost node in the graph panel) was selected. Nine possible next states (including the root itself) are highlighted, while the data panel visualises the initial state, that is the graph $G_{t=t'}$. This graph features two edges labelled by f and attached by their result node $n2$. By synchronising a rewrite of the fist edge by the axiom of figure 3 with a rewrite of the second edge by the axiom $\mathbf{f}(u\,x\,y) \xrightarrow{u,\overline{f},x\,y} \emptyset$ (an instance of the second scheme above), we obtain a transition to the graph depicted in figure 4. One more transition, where the two g-labelled and the two z-labelled edges synchronise in parallel, and we reach a final state, that is a state with a unique output edge to itself. All graphs associated with such a state are isomorphic to the one depicted in figure 5, representing both the unified term

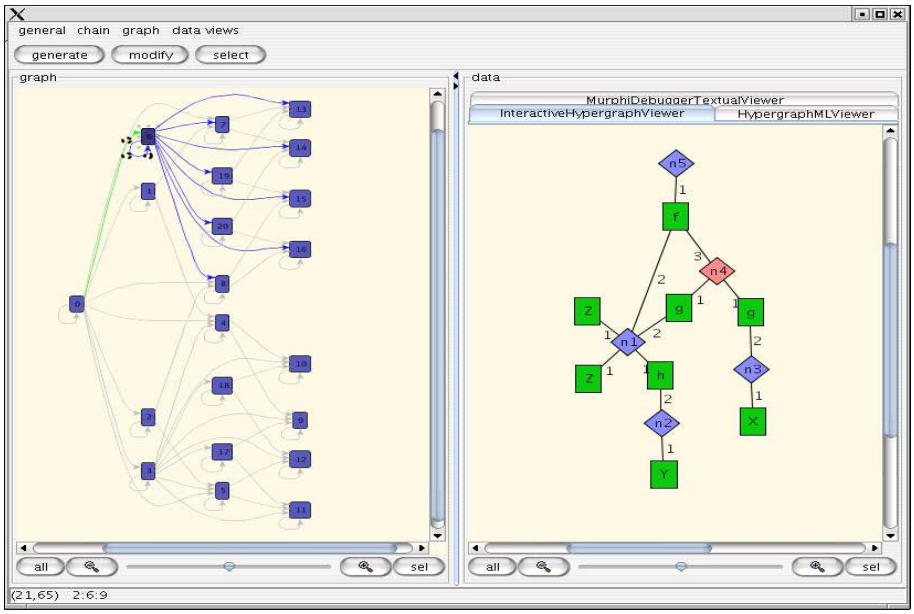

Figure 4. second step of computation.

$f(h(y), g(h(y)))$ and the unifier ρ mapping x and z to $h(y)$. How to extract the unifier from a final graph is explained in [4] and implemented in `shegrave` by means of a viewer which can be selected by clicking the button at the top of the data panel.

5 Leader election

In this section we show how to program the *Leader Election Protocol* using SG and SHE. Let us recall the protocol (as presented in [1]). The aim of this protocol is to to choose, among a set of processors, one as the *leader*. Each processor is labeled with a unique number and they are organized as a *ring* so that every processor is adjacent to two neighbors. Processors can communicate by sending messages: every processor can receive messages from its left neighbor and send messages to its right neighbor. Communication is safe: once a message is sent we are assured that it will be delivered. The ring is *asynchronous*: no assumptions are made on the scheduling of the processors and also on the time that it takes for a message to be read. A protocol that solves the problem is the following: first every processor sends a message containing its label then it starts receiving messages. The behavior of a processor p when a message m is read depends on the message's content: if $p > m$ then the message is discarded, instead if $p < m$ the message is forwarded while if $p = m$ then p becomes the leader.
Now we show how to program this protocol in SG. Notice

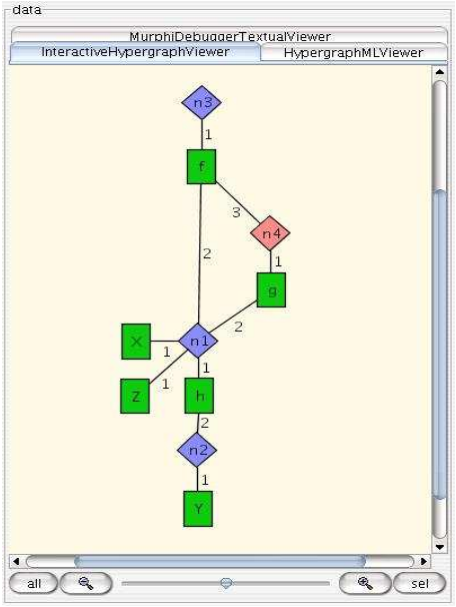

Figure 5. the result of unification.

that besides the protocol also the *computational model* is modeled (e.g. topology, asynchrony). A processor with label p is represented by an edge p attached to two different nodes: $p(x\,y)$. Node x is shared by p and its left neighbor, while y connects p to its right neighbor. If a processor p has not yet sent its first message $p^s(x\,y)$ is used instead of $p(x\,y)$. A message m is represented by an edge with a single node $m(x)$. Since every communication channel is shared by only two processors, every node is restricted. So, any given ring with processors (from left to right) $p_1 p_2 \dots p_n$ is represented by a graph $\nu x_1 x_2 \dots x_n . p_1(x_1\,x_2) | p_2(x_2\,x_3) | \cdots | p_n(x_n\,x_1)$. The following productions, with the addition of all the identities realize the protocol:

$$p^s(x\,y) \longrightarrow p(x\,y) | p(y) \tag{1}$$

$$p(x\,y) \xrightarrow{(x,\overline{m})} p(x\,y) | m(y) \qquad \text{if } m > p \tag{2}$$

$$p(x\,y) \xrightarrow{(x,\overline{m})} p(x\,y) \qquad \text{if } m < p \tag{3}$$

$$p(x\,y) \xrightarrow{(x,\overline{m})} L(x\,y) \qquad \text{if } m = p \tag{4}$$

$$m(x) \xrightarrow{(x,m)} \emptyset \tag{5}$$

$$L(x\,y) \xrightarrow{(x,\overline{m})} L(x\,y) \tag{6}$$

Production 1 is used by a processor to send its first message, which contains its label. Production from 2 to 4 are used to read a message: with 2 a message is read and discarded, with 3 it is forwarded while using 4 a processor receives a message containing its own id and it becomes leader. Messages have only one production (5), which is applied when they are received. Production 6 is used by a processor that has become leader to continue receiving messages. Those messages are completely ignored, actually this production is provided only to satisfy the safety condition on communication, which guarantees that every message is received.

Programming this protocol in SHE is straightforward. Hence, given an arbitrary ring we can check that our protocol is correct, at least for that specific instance, simply by generating the state graph and checking that every computation ends in a graph which has exactly one leader. This can be done either by hand or by building a simple node selector that shows all the final states. In a similar way we can check that the state graph does not contain any cycles. An applet showing the protocol running is avaible at HTTP://BRIANTB.UNIXCAB.ORG/SHETEMP/ in the download section. Here we also provide a proof of correctness for the protocol as we have programmed it in SG.

Let us call *idle* a transition of the form $G \xrightarrow{\emptyset} G$. Also call *idle* a (possibly infinite) computation composed only by idle transitions. A *run* of a graph G is a computation $G \xrightarrow{\Lambda_1} \cdots \xrightarrow{\Lambda_n} F$ in which every transition is non-idle and such that every computation starting from F is idle. Let us also denote by $\mathfrak{R}^s_n$ an arbitrary ring with n processors all in the initial state, all of them equipped with a different label:

$$\mathfrak{R}^s_n \stackrel{\text{def}}{=} \nu\, x_1 \dots x_n .$$

$$(p^s_{id_1}(x_1\,x_2) \,|\, p^s_{id_2}(x_2\,x_3) \,|\, \cdots \,|\, p^s_{id_n}(x_n\,x_1))$$

. The following theorem holds.

Theorem 1. *Let $\mathfrak{R}^s_n \xrightarrow{\Lambda_1} \cdots \xrightarrow{\Lambda_k} E$ be a run of $\mathfrak{R}^s_n$. Then E is of the following form:*

$$\nu x_1 \dots x_n .$$
$$p_{id_1}(x_1\,x_2) \,|\, \cdots \,|\, p_{id_{j-1}}(x_{j-1}\,x_j) \,|$$
$$L(x_j\,x_{j+1}) \,|\, p_{id_{j+1}}(x_{j+1}\,x_{j+2}) \,|\, \cdots \,|\, p_{id_n}(x_n\,x_1)$$

.

Proof. Observe first that in E there can be neither a message that has not been read nor a processor still in the initial state, otherwise from E we could do a non $idle$ transition to a different graph E' (respectively by reading and by sending a message). Note also that messages are never lost, that is if in a transition a production like 5 is used then during the same transition also the corresponding processor production (i.e. a production that issues the co-action $\overline{m}$ on the same node) must be used, since all nodes are restricted and so every action has to be synchronised. For the very same reason a processor can read a message only if it is actually there. Now observe that if a processor p receives a message containing its own label, then this message has already been read by all the other processors. This is because p is the only one who can generate a message with its own label (since labels are assumed to be distinct) and because when messages are forwarded they are not altered, as we can see from the productions. So a message sent by p before coming back to the processor must be forwarded through all the ring. But if a message p is forwarded by all processors then p is the greatest label across all the ring. So there can be only one processor with such a label. □

Note also that every *run* is finite as the following lemma shows.

Lemma 1. *Every computation starting from $\mathfrak{R}^s_n$ and such that every transition in it is non idle is of finite length.*

Proof. It is easy to see that in every non idle transition at least one message is either sent for the first time or read (and then discarded or forwarded). The number of transitions in which a fresh message is generated is at most n. Each message can not be read more than n times. So the length of those computations is bounded by $n \times (n+1)$ □

6 Conclusions

Even if SHE is fully functional and, as we have shown, can be used for quite different applications, there are still many features that we would like to add. First of all we would like to support the full version of SG so as to allow unsynchronised actions, which are interesting when analysing subsystems in isolation. Another interesting issue is to fully integrate Murphi in SHE in a way that makes it possible and easy to take advantage of all its features: up to now inside SHE Murphi is used mainly as an engine that generates the state graph. If we want to check that some properties are preserved *during* the generation of the state graph and not after it has been fully generated, we have to hard code them by hand inside the Murphi file. So it would be much easier if we could specify, together with the starting graph, the properties in which we are interested and then automatically generate a Murphi executable capable of testing their validity. Another quite interesting evolution of SHE consists in turning it into a common workbench for different process calculi. This is a very natural development since the programming language of SHE is SG. SG has been proved capable of providing models for many process calculi, in an easy intuitive way. So using SG as an intermediate language we would be able to turn SHE into an interpreter for virtually any interesting process calculus. In addition, being the architecture of SHE very modular, implementing this extension should prove to be a not so difficult task. The main effort would be to write, for each process calculus supported, a module implementing the translation from that calculus into SG. Finally, a promising development of SHE is tied to a recent advance in SG theory. In fact the authors have formulated a notion of behavioural equivalence for SG, which essentially allows to equate graphs showing the same behaviour. If we consider SG as a declarative programming language, this notion makes it possible reasoning on programs equivalence, allowing for instance to tell whether a particular implementation of an abstract specific is correct or not. Our intention is to add to SHE a module that deals with equivalence, thus allowing to automatically check programs for equivalence.

References

[1] H. Attiya and J. Welch, *Distributed computing*, McGraw-Hill, 1998.

[2] Brandes, Eiglsperger, Herman, Himsolt, and Marshall, *GraphML progress report (structured layer proposal)*, GDRAWING: Conference on Graph Drawing (GD), 2001.

[3] P. Cenciarelli, I. Talamo, and A. Tiberi, *Ambient Graph Rewriting*, Proceedings of WRLA'04. To appear in Elsevier ENTCS, 2004.

[4] P. Cenciarelli and A. Tiberi, *Rational Unification in 28 Characters*, Proceedings of 2nd International Workshop on Term Graph Rewriting (TERMGRAPH'04). To appear in Elsevier ENTCS, 2004.

[5] ———, *Synchronising Graphs*, Submitted, 2004.

[6] R. Cleaveland, J. Parrow, and B. Steffen, *The Concurrency Workbench*, Springer LNCS **407** (1989), 24–37.

[7] P. Degano and U. Montanari, *A model of distributrd systems based on graph rewriting*, Journal of the ACM **34** (1987), 411–449.

[8] David L. Dill, *The murphi verification system*, April 08 1998.

[9] G. Ferrari, U. Montanari, and E. Tuosto, *A LTS semantics of ambients via graph synchronization with mobility*, Proc.ITCS 01, Springer LNCS 2202 (2001).

[10] G.J. Holzmann, *The Model Checker Spin*, IEEE Trans. on Software Engineering **23** (1997), no. 5, 279–295.

[11] A. Robinson, *A Machine oriented Logic Based on the resolution principle*, Journal of the ACM **12** (1965), 23–41.