43rd IEEE Conference on Decision and Control
December 14-17, 2004
Atlantis, Paradise Island, Bahamas

Distributing Synchronous Systems with Modular Structure

Marco Zennaro and Raja Sengupta, {zennaro, raja}@path.berkeley.edu

Abstract— **Synchronous programs were introduced to simplify the development of reactive systems hiding the complexity and indeterminism of the interleaving while taking full advantage of possible concurrency. The introduction of communication networks enabled the creation of distributed systems presenting the programmer with a new burden of interleaving and non determinism due to the asynchronous communication medium. Again this complexity should be hidden from the user while taking full advantage of the possible concurrency to improve performance. Many algorithms for the automatic distributions of synchronous programs have been proposed so far, but they are not suitable for large scale system because they do not preserve the compositionality of the original code: the modularity of the synchronous program is lost. As a result the subsystems are not re-usable and a small local change results in the recompilation and re-distribution of the overall system. This solution is cumbersome and unpractical in many real-world applications. In this paper we introduce an algorithm for the distribution of synchronous programs that preserves the modularity and allows separate compilation and subsystem re-use.**

I. INTRODUCTION

In this paper we propose an algorithm for the modular compilation and automatic distribution of SIMULINK-like control programs across networks.

The synchronous paradigm was introduced in order to simplify the programming of reactive systems, hiding from the user the complexity of interleaving and its associated non determinism [1]-[2]. The compiler takes care of translating the synchronous system into sequential (asynchronous) code while preserving its semantic [2]-[3]. Synchronous programming languages like ESTEREL [4], LUSTRE [5], SIGNAL [6], or SIMULINK are modular and compositional. This is essential for the programming of large control systems.

Communication networks enable systems to be distributed, enhancing both concurrency and non-determinism. In the synchronous philosophy, the resulting complexity should be hidden from the user and automatically taken care of by the compiler. This is now an active field of research. [8]-[9] propose algorithms to distribute particular subsets of ESTEREL programs, starting with a single synchronous program and splitting it into synchronous subsystems intercommunicating through an asynchronous medium creating what is called a Globally Asynchronous Locally Synchronous (GALS) system [10]. This approach preserves the synchronous semantics but does not maintain or exploit the modular structure in the original synchronous program. Consequently, modification to one module of the synchronous program may require re-compilation and re-distribution of the entire system.

We try to achieve the same objectives while retaining any modular structure in the synchronous program in its asynchronous, semantic preserving, equivalent. Our aim is a distribution method in which any modification to a module of the synchronous program will only require recompilation of the altered module.

[10] proves such a mapping to GALS, preserving modularity, exists for a particular class of synchronous systems. However, no algorithm computing on a finite representation of synchronous systems is given.

In this paper we give such an algorithm. Our approach is to develop a formalism for a synchronous program and one for an asynchronous program. Our synchronous formalism is like SIMULINK and less general than ESTEREL. The asynchronous formalism is like the I/O automata of [11]. We define a synchronous and asynchronous composition operator. The synchronous composition operator is SIMULINK-like. The asynchronous composition operator is CSP-style rendezvous [12]. We present an algorithm to implement a synchronous program into an asynchronous one and prove the implementation map preserves the synchronous semantics in the sense of [10]. The main result is that the implementation is a monomorphism with respect to the synchronous and asynchronous compositions. The monomorphism is our argument that a local change can be handled locally and that a subsystem can be re-used in different systems. The arguments of this paper are mathematical. A future paper will describe the translation of this mathematics into software.

Sections II and III of this paper describe our synchronous and asynchronous system formalisms. Section IV formulates the modular compilation and distribution problem. Section V is about implementing a synchronous component as an asynchronous one preserving its synchronous semantics. Section VI gives the mathematical argument about modular compilation and re-distribution.

II. SYNCHRONOUS SYSTEMS

Several synchronous systems formalisms exists in the literature. The basic idea behind all of them is of a systems evolving through discrete steps. At every step all the variables are updated and they do not change values until the next step is taken.

A. STS and STS*

The Synchronous Transition System (STS) formalism, introduced by Manna and Pnueli [13] models a synchronous

The work was supported by Office of Naval Research (AINS), grant N00014-03-C-0187 and SPO 016671-004

Authors are with the Center for Collaborative Control of Unmanned Vehicles, University of California, 2105 Bancroft way, Berkeley CA 94704

system s as a couple (P_s, B_s) where P_s is the set of the I/O ports and state variables of the system and B_s is the set of the traces admitted by the system; a trace is an infinite sequence of states and a state is a valuation of all the element of P_s. If P is a set of ports, we denote by $\sigma(P)$ a valuation of the ports in the set P and by $\Lambda(P)$ the set of the possible valuations of the ports in P.

This lightweight formalism is not finite, hence it cannot be input to an algorithm. In this paper we propose a different form of STS, the STS* formalism, that keeps its simplicity, it is finite and easier to relate to SIMULINK. We define a STS* as the tuple $(P_I, P_O, P_S, I_0, \psi_O, \psi_S, \prec)$ where P_I is the finite set of input ports of the system; P_O is the finite set of output ports of the system; P_S is the finite set of state variable of the system; I_0 are the initial valuation of the state variables in P_S; Ψ_O is a set of computable functions (one for every output port) used to compute the system outputs, having the following signature:

$$\forall \psi \in \Psi_O \,.\, \psi : \Lambda(P_I^\psi) \times \Lambda(P_S^\psi) \rightarrow \Lambda(P_O^\psi)$$

where $P_I^\psi \subseteq P_I$, $P_S^\psi \subseteq P_S$, and $P_O^\psi \in P_O$. ψ^p denotes the output function with output port p; Ψ_S is a set of computable functions (one for every state variable) used to compute the next system state. The following holds:

$$\forall \psi \in \Psi_S \,.\, \psi : \Lambda(P_I^\psi) \times \Lambda(P_S^\psi) \rightarrow \Lambda(P_S^{'\psi}).$$

where $P_I^\psi \subseteq P_I$, $P_S^\psi \subseteq P_S$, and $P_S^{'\psi} \in P_S$. We denote the unique function for which $P_S^{'\psi} = p$ with ψ^p; $\prec$ is an acyclic partial order over $P_I \cup P_O \cup P_S$ expressing the causality relation between input, outputs and state variables (e.g. if the output O_i is the sum of I_1 and I_2 then O_i depends upon I_1 and I_2, written $I_1, I_2 \prec O_i$). The following holds:

$$\begin{aligned}(\alpha,\beta) \in \prec \quad \Leftrightarrow \quad & \exists \psi \in \Psi_O \,.\, \alpha \in P_I^\psi \wedge \beta \in P_O^\psi \vee \\ & \exists \psi \in \Psi_S \,.\, \alpha \in P_I^\psi \wedge \beta \in P_S^\psi\end{aligned} \tag{1}$$

B. STS semantics

The semantic is given in terms of traces. A trace t is an infinite sequence of valuations of P_I,P_O and P_S. The i^{th} vector of valuations in a trace t is denoted by t_i. We denote with $t|P$ the projection of the trace t over the set of ports and variables P. Given a trace t, we say that t_i satisfies the system s, denoted $s \models t_i$, if the following holds:

$$\begin{aligned}s \models t_i \quad \Leftrightarrow \quad & i = 0 \Rightarrow \forall p \in P_S \,.\, t_0|p = I_0|p \wedge \\ & \forall p \in P_O . t_i|p = \psi_0^p(t_i|(P_I^{\psi_0^p} \cup P_S^{\psi_0^p})) \wedge \\ & \forall p \in P_S . t_{i+1}|p = \psi_S^p(t_i|(P_I^{\psi_S^p} \cup P_S^{\psi_S^p}))\end{aligned}$$

We say that an STS* system s admits a trace t (or that trace t satisfies the system s), written $s \models t$, iff:

$$s \models t \Leftrightarrow \forall i \in \mathbb{N}\ s \models t_i$$

If $\prec$ is acyclic the state t_{i+1} can be computed from the state t_i and the inputs, while it is not always possible if there is a cycle. Some authors have looked for fixed-point solution [14], while others assumed out the case [9], as we do.

C. Compatible STS* systems composition

We now define a composition for STS*. A complex system is created composing subsystems. Not all systems can be composed. Given two STS systems s=($P_I^s, P_O^s, P_S^s, I_0^s, \psi_O^s, \psi_S^s, \prec^s$) and t=(P_I^t, P_O^t, P_S^t, I_0^t, ψ_O^t, $\psi_S^t, \prec^t$), their composition, denoded with $s \times_{STS} t$ is defined iff:

$$P_O^s \cap P_O^t = \emptyset \ \wedge\ P_S^s \cap P_S^t = \emptyset \wedge\ \prec^s \cup \prec^t \text{ is acyclic.}$$

The first condition avoids output racing conditions and the consequent non-determinism, the second ensures state variables are local to the component, the third that the composed system does not have cyclic causal variables dependencies. If the three conditions hold, the systems are said to be *compatible* and their composition $\times_{STS}$ is defined as follows:

- $P_S^{s\times t} = P_S^s \cup P_S^t$, $P_I^{s\times t} = P_I^s \cup P_I^t$, $P_O^{s\times t} = P_O^s \cup P_O^t$;
- $I^{0s\times t} = I_0^s \cup I_0^t$, $\prec^{s\times t} = \prec^s \cup \prec^t$;
- $\psi_O^{s\times t} = \psi_O^s \cup \psi_O^t$, $\psi_S^{s\times t} = \psi_S^s \cup \psi_S^t$

Therefore $\times_{STS}$ is a partial function over the STS* set.
It is easy to translate a Simulink program into an STS*: the semantic of the Simulink composition is maintained by $\times_{STS}$,and every block can be described by its I/O ports, state variables, and their update functions.

Next we state two lemmas. The lemmas assert our STS* formalism has the usual properties of other formalisms in the literature.

Lemma 1: (STS*, $\times_{STS}$) is a commutative monoid, with the identity element being the empty STS*.

Proof: Follows from the associativity and commutativity of the union operator and by the fact that the identity element of the union operator is the empty set.

Lemma 2: Given two STS* s_1 and s_2, $s_1 \times_{STS} s_2 \models t \Leftrightarrow s_1 \models t|(P_I^{s_1} \cup P_O^{s_1} \cup P_S^{s_1}) \wedge s_2 \models t|(P_I^{s_2} \cup P_O^{s_2} \cup P_S^{s_2})$

Proof: The proof is by contradiction. Let's assume that:

$$s_1 \times_{STS} s_2 \models t \wedge \neg s_1 \models t|(P_I^{s_1} \cup P_O^{s_1} \cup P_S^{s_1})$$

(the other cases are symmetrical). If this is the case than there is a minimal $i \in \mathbb{N}$ for which:

$$s_1 \times_{STS} s_2 \models t_i \wedge \neg s_1 \models t_i|(P_I^{s_1} \cup P_O^{s_1} \cup P_S^{s_1}).$$

Then there must exist at least one p in $P_O^{s_1} \cup P_S^{s_1}$ for which:

$$\begin{aligned}p \in P_O \Rightarrow t_i|p \neq \psi^p(t_i|(P_I^{\psi_0^p} \cup P_S^{\psi_0^p})) \\ p \in P_S \Rightarrow t_i|p \neq \psi^p(t_{i-1}|(P_I^{\psi_0^p} \cup P_S^{\psi_0^p})).\end{aligned}$$

Pick one such p that is minimal with respect to $\prec_{s_1}$. Denote it by p_0 and assume it is an output port (the case for a state variable is identical). Since p_0 and i are chosen to be minimal

$$\forall p \in P_I^{\psi_{p_0}} \cup P_S^{\psi_{p_0}} t_i|p = \psi^p(P_I^{\psi^p} \cup P_S^{\psi^p})$$

Now by the definition of STS* composition and by (1):

$$t_i|p_0 = \psi^{p_0}(t_i|(P_I^{\psi_0^p} \cup P_S^{\psi_0^p})$$

But this contradicts the hyphothesis, hence the lemma is proved. Q.E.D.

III. ASYNCHRONOUS SYSTEMS

Several asynchronous system formalisms exists in the literature. One of them is the asynchronous version of STS, called the Asynchronous Transition System (ATS) model, as introduced by Benvenieste in [1]. In ATS an asynchronous system is a couple (P_a, B_a) where P_a is the set of I/O ports and B_a the set of the possible behaviors. A behavior is an infinite sequence of valuations and a valuation is a couple (port number, value). Again the simplicity of the model makes it easy to handle it theoretically, but we seek a finite formalism to be input to an algorithm.

Instead we use Reactive Automata (RA), labeled finite automata augmented with variables [11] communicating through ports (modelled as shared variables) in a CSP manner [12]. Formally an RA is a tuple (L, l_0, V, V_0, P, T) where L is a finite set of locations of the automaton; $l_0 \in L$ is the initial location; V is a finite set of variables read and written only by the RA; V_0: is the initial value of the variables; P is a finite set of communication ports, modelled as environmental variables, read and written by any RA; T is a finite set of labeled transitions of the form $(l_i, l_f, (c, A))$ where $l_i, l_f \in L$, c is a boolean condition over V or $?p(v)$, $p \in P$ and $v \in V$. A is a sequence of actions defined by the following grammar:
$A \rightarrow nil$
$A \rightarrow !p(v)$ where $p \in P$ and $v \in V$
$A \rightarrow v := f(V_1); A$ where $v \in V$, $V_1 \subseteq V, f \in \mathbb{F}(V_1)$
We denoted with $\mathbb{F}(V_1)$ is the set of computable functions $\Sigma(V_1) \rightarrow \Sigma(v)$. An example of such an automaton is given in figure 1.

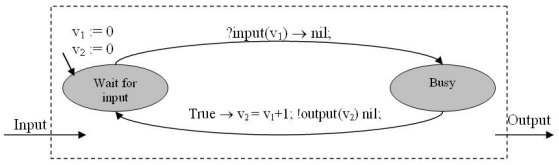

Fig. 1. A simple reactive automaton

The semantic of an RA is in terms of runs and traces. A run is an infinite sequence of tuples (location, variables valuation, transition). A trace is a tuple, where each element of the tuple is an infinite sequence of variable valuations. For example the following is a representation of (the initial part) a run of the RA in figure 1 for the input "1":

$$(W,0,0) \overset{?in(v_1)}{\rightarrow} (B,1,0) \overset{v_2:=v_1+1;?out(v_2)}{\rightarrow} (W,1,2)$$

Where W is the *wait for input* location, B is the *busy* location and the second and third numbers are valuations of v_1 and v_2. Mathematically we represent the run segment above by:

$$< (W,0,0,?in(v_1)), (B,1,0,v_2 := v_1 + 1; ?out(v_2), ... >$$

The i^{th} tuple in a run r is denoted by r_i. The i^{th} valuation of a variable v in a trace t is denoted by t_i^v. Given a run, the associated trace can be computed by examining the updates of every single variable in the run.

Given a run r, we say that the tuple r_i satisfies a RA w, denoted $w \models r_i$ iff the following holds:

$$\begin{array}{rcl} w \models r_i & \Leftrightarrow & i = 0 \Rightarrow r_0|location = l_0 \wedge \\ & & i = 0 \Rightarrow r_0|V = V_0 \wedge \\ & & \exists (s, s', (c, a)) \in T \begin{array}{l} r_{i+1}|state = s' \wedge \\ r_i|V \models c \wedge \\ r_{i+1}|V = \\ \quad act(a, c, r_i|V) \end{array} \end{array}$$

where the function act is defined as follows:

$$act(a,c,\sigma) = \begin{cases} act(a, true, \sigma') & \text{if } c = ?p(v') \\ & \forall v \neq v'.\sigma'|v = \sigma|v \\ \sigma & \text{if } a = nil \ \wedge c \neq ?p(v) \\ \sigma & if\ a = !p(v) \wedge c \neq ?p(v) \\ act(a_1, c, \sigma') & if\ a = v' := f(V); a_1 \\ & \wedge\ c \neq ?p(v) \wedge \\ & \forall v \neq v'.(\sigma'|v = \sigma|v \\ & \wedge\ \sigma'|v' = f(V)) \end{cases}$$

where σ and σ' are valuation of the variables in V.

A run r satisfies a RA w, denoted $w \models r$ iff each one of its tuple does it (i.e. $\forall i \in \mathbb{N}\ w \models r_i$). A trace t satisfies a RA w, denoted $w \models t$ iff there is a run r such that $w \models r$ and t is associated to r.

IV. PROBLEM STATEMENT

We now formally define our problem. Figure 3 illustrates the research program. First we need to find a way to associate RA and STS* traces, i.e. we need a trace map $\chi : \mathbb{T}_{RA} \rightarrow \mathbb{T}_{STS*}$ where $\mathbb{T}_{RA}$ and $\mathbb{T}_{STS}$ are the set of traces of STS and RA respectively. In [10] the following definition of an invertible map χ is given:

$$t' = \chi(t) \Leftrightarrow \forall i \in \mathbb{N}\ \forall v_i \in V\ .\ t_i^{v_i} = t_i'|v_i$$

First we want to implement STS* as RA while preserving the synchronous semantic, i.e. we need to find an implementation map ϕ :STS*$\rightarrow$RA such that for all STS* s and RA r, the following holds:

$$(r = \phi(s)\ \wedge r \models t) \Leftrightarrow s \models \chi(t) \qquad (2)$$

It has been proved in [10] that for the set of *endochronous* programs such a ϕ exists. In section V we define a ϕ for the class of STS*.

So far we have just obtained what a Simulink compiler does, or what is done in [2]. We can now formulate our problem (like [10]) as follows: we seek a composition operator $\times_{RA}$ such that, for any two STS* s_1 and s_2 and

RA r_1 and r_2, the following holds:

$$r_1 = \phi(s_1) \wedge r_2 = \phi(s_2) \wedge r_1 \times_{RA} r_2 \models t \Leftrightarrow s_1 \times_{STS} s_2 \models \chi(t) \quad (3)$$

If this holds and if the composition operator $\times_{RA}$ can be implemented across a network than this constitutes a way to distribute the synchronous system $s_1 \times_{STS} s_2$ across a network while preserving its synchronous semantic. It has been proved in [10] that when the pair (s_1, s_2) is *isochronous* than such an operator exists. In section VI we define an operator $\times_{RA}$ for which we prove that property (3) holds if the two synchronous system are *compatible* (as defined in section II). In this we achieved the goal of distributing synchronous programs while preserving modularity.
Because $\times_{RA}$ satisfies (2), a local change in s_1, requires only local recompilation (using ϕ) while the rest of the system is kept unchanged. Once compiled, an RA $r = \phi(s)$ can be used right away in a different system without any global recompilation and distribution. This is possible because the modular structure of the synchronous system is preserved even as it is translated into an asynchronous system.

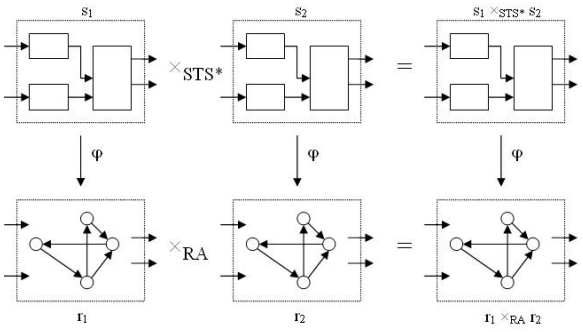

Fig. 2. A graphical representation of property (3)

V. IMPLEMENTATION OF SYNCHRONOUS SYSTEMS

We now focus on the implementation map ϕ between STS* and RA. We give an algorithm for ϕ and we prove that it satisfies property (2).

A. Implementation algorithm

Given an STS* s we compute its implementation, the RA $r = \phi(s)$ as follows:

Algorithm *CG* (Compute Graph)
Input: *A partial order $\prec$ over a set V, the set V, and two labels root, leaf*
Output: *A graph (Nodes, Edges)*

1 $Nodes = root, leaf$
2 $Edges = \emptyset$
3 $counter = 0$
4 Add $root$ and $leaf$ to $Nodes$
5 $\forall$ linearization $w = (w_1, w_2, ..., w_m)$ of $\prec$ in V do
6 $pointer = root$
7 For all $i \in [0, m]$ do
8 if $(pointer, n, w_i) \notin Edges$ do $pointer = n$
9 else do
10 add $n_{counter}$ to $Nodes$
11 add $(pointer, n_{counter}, w_i)$ to $Edges$
12 $pointer = n_{counter}$
13 $counter++$
14 od
15 od
16 od
17 Replace the sinks in $Nodes$ and $Edges$ with $leaf$

Algorithm *Implement*
Input: *An STS* s = $(P_I, P_O, P_S, I_0, \psi_O, \psi_S, \prec)$*
Output: *An RA r = $\phi(s)$ = (L, l_0, V, V_0, P, T)*

1 $P = \{p_j | j \in P_I\} \cup \{p_j | j \in P_O\}$
2 $V = \{v_i | i \in (P_I \cup P_O \cup P_S)\}$
3 $V_0 = I_0$
4 $L_0 = l_{root}$
5 $L = \{l_{root}, l_{leaf}\}$
6 $(N, E) = CG(\prec |(P_I \cup P_O), (P_I \cup P_O), root, leaf)$
7 For all $n \in N$ add l_n in L
8 For all $(n, n', v) \in E$ do
9 if $v \in P_I$ then do
10 add $(l_n, l_{n'}, (?P_i(v), nil))$ to T
11 od
12 if $v \in P_O$ then do
13 add $(l_n, l_{n'},$
$(true, v_i = \psi(V|P_I^i \cup P_S^i); !P_i(v_i)))$ to T
14 od
15 od
16 $(N, E) = CG(\prec |(P_S), (P_S), leaf, root)$
17 For all $n \in N$ add l_n in L
18 For all $(n, n', v) \in E$ do
19 add $(l_n, l_{n'}, (true, v_i = \psi()!P_i(v_i)))$ to T
20 od

The algorithm is guaranted to terminate for every STS*. All the for loops terminate in finitely many steps because the set of variables and ports of an STS* is finite.

B. Correctness proof

We start our proof proving the following lemmas:

Lemma 3: $ComputeGraph$ produces an acyclic graph with source $root$ and sink $leaf$. Every path in the graph from source to sink has one and only one edge labelled with an element of V. Moreover if $v' \prec v$ and $\{v, v'\} \subset V$ then the edge labelled v' appears before the one labelled v in every path.

Proof: Every time an edge is added, it does not create a loop because it connects an existing node to a new one. Line 17 does not create loop since it flattens all the leafs (i.e. sinks) into a single sink. By construction every path corresponds

to a linearization of $\prec$ in V, and is therefore acyclic. Hence an element v' of V appears as a label only once and it appears before all the v for which $(v', v) \in \prec$. Q.E.D.

Lemma 4: For all r in $\phi[STS*]$ and every run t of r, t visits the location l_{leaf} infinitely often.

Proof: The automata generated by the algorithm $Implement$ are obtained linking two graphs generated by $ComputeGraph$. so that the source of one is the sink of the other. The only nodes that are shared by the two graphs are $root$ and $leaf$. Thus every path starting from l_{root} ends in l_{leaf} since $leaf$ is the sink of the first graph and there are finitely many states in it. Similarly the state l_{root} is encountered after finitely many states, because it is a sink of the second graph. Hence, since a run is infinite, the state l_{leaf} is encountered infinitely often. Q.E.D.

From lemma 4 we see that any run r = $< r_0, r_1, r_2, ... >$ of an RA in $\phi[STS*]$ has an infinite subsequence $< r_{i_0}, r_{i_1}, r_{i_2}, ... >$ such that $\forall k \in \mathbb{N}$- 0 $r_{i_k}|S = l_{leaf}$ and $(\forall k \in \mathbb{N}\ r_i \neq r_{i_k}) \Rightarrow r_i|S \neq l_{leaf}$. Thus we can write r equivalently as r = $< u_0, u_1, u_2, ... >$ where $u_0 =< r_0 >, u_1 =< r_1, ..., r_{i_1-1} >, u_2 =< r_{i_1}, ..., r_{i_2-1} >$ and so on. We call this u_i's *cycles*. We can also define the function $cycle(r, n)$, for r a run and $n \in \mathbb{N}$ as the valuation of V at the n^{th} visit to l_{leaf}, i.e. $cycle(r, n) = r_i|V$

Lemma 5: In every cycle all the variables of the RA obtained using the implementation algorithm are valuated once and only once. If $v' \prec v$ in the original STS* then v' is valuated before v in every cicle.

Proof: Follows from lemma 3 and the definition of $Implement$.

Lemma 6: For a given run r of an RA $\in \phi(STS*)$, let t be the associated trace. Then the following holds: $\forall i \in \mathbb{N}\ \forall v_i \in V\ .\ t_i^{v_i} = cycle(r, i)|v_i$

Proof: By lemma 5 in every cycle a variable is evaluated once and only once before hitting the sink l_{leaf}. When a run hits the location l_{leaf} for the i^{th} time, all the variables have been evaluated exactly i times. Q.E.D.

The first theorem stated below asserts algorithm *Implement* constructs an RA implementation of an STS* while preserving its semantics in the sense of χ.

Theorem 1: ϕ defined by algorithms $Implement$ that computes ϕ satisfies property (2)

Proof: Assume that the theorem does not hold. Then the following must hold:

$$\exists s \in STS*, \exists r = \phi(s) \in RA, \exists t \in \Gamma\ .\ r \models t \wedge \neg s \models \chi(t)$$

where Γ is the set of traces of r. For the previous statement to hold there must be at least one i and a variable v for which:

$$\chi(t)_i|v \neq \psi^v(\chi(t)_i|P_i^{\psi^v} \cup P_s^{\psi^v}) \tag{4}$$

Select the smallest i for which (4) holds. Similarly fix one of the minimal variables v wrt $\prec$. Hence, by the minimality of v the following must hold:

$$\forall v' \prec v\ .\ \chi(t)_i|v' = \psi^{v'}(\chi(t)_i|P_i^{\psi^{v'}} \cup P_s^{\psi^{v'}}) = t_i^{v'}. \tag{5}$$

By lemma 6:

$$t_i^{v'} = V|v' \text{ where } V = cycle(r, i).$$

So that from (5) it follows that:

$$\forall v' \prec v\ .\ \chi(t)_i|v' = V|v' \text{ where } V = cycle(r, i). \tag{6}$$

So that from (6) and (1) we get:

$$\chi(t)_i|(P_i^{\psi^{v'}} \cup P_s^{\psi^{v'}}) = V|(P_i^{\psi^{v'}} \cup P_s^{\psi^{v'}}) \tag{7}$$

So that:

$$\begin{aligned} \chi(t)_i|v &= t_i^v && \text{[by definition of } \chi\text{]} \\ &= \psi^v(V|(P_i^{\psi^{v'}} \cup P_s^{\psi^{v'}}) = && \text{[by lemma 5]} \\ &= \psi^v(\chi(t_i)|P_i^{\psi^{v'}} \cup P_s^{\psi^{v'}}) && \text{[by (7)]} \end{aligned}$$

But this contradicts (4). Hence the theorem 1 holds. Q.E.D.

VI. MODULAR DISTRIBUTION OF SYNCHRONOUS SYSTEMS

We have proved in the previous section that there is a map ϕ between STS* and RA satisfying property (2). We now introduce a composition operator $\times_{RA}$ for which property (2) holds and we prove that ϕ is a monomorphism between (STS*, $\times_{STS}$) and (RA, $\times_{RA}$)

A. Rendezvous composition

We define $\times_{RA}$ as a rendezvous style composition (as done for CSP in [12]). If two reactive automata are communicating through a port p, the writer RA writes on it only when the reader is reading it. This is a blocking composition. In other words given two RA r_1 and r_2 the following holds:

$$\begin{aligned} &r_1 \times r_2 \models r \Rightarrow r_1 \models r|V_{r_1} \wedge r_2 \models r|V_{r_2} \wedge \\ &\forall i, j, k, k'\ r_i|_A = !p_j(v'_k) \Leftrightarrow (r_{i+1}|_C = ?p_j(v_k) \wedge \\ &\forall v \in V v \neq v_k\ r_i|v = r_{i+1}|v \wedge r_{i+1}|v_k = r_i|v_{k'} \wedge \\ &\forall i, j, k, k'\ r_{i+1}|_A = ?p_j(v_k) \Leftrightarrow (r_i|_C = !p_j(v'_k) \wedge \\ &\forall v \in V v \neq v_k\ r_i|v = r_{i+1}|v \wedge r_{i+1}|v_k = r_i|v_{k'} \end{aligned} \tag{8}$$

where V_{r_1} and V_{r_2} are the variables of r_1 and r_2 respectively. From (8) it follows that:

$$\forall t \in \Gamma\ .\ r_1 \times_{RA} r_2 \models t \Rightarrow r_1 \models t|V_{r_1} \wedge r_2 \models t|V_{r_2} \tag{9}$$

where Γ is the set of traces of $r_1 \times_{RA} r_2$.

In general two different RA do not share the same notion of time. But, if we are using $\times_{RA}$, then we can claim the following: if a variable v in one RA is valuated before writing on a port P and on the other side a variable v' is valuated after reading from P then we can be sure that v has ben valuated before v'. For the class of reactive automata implementying an STS*, i.e. $\psi[STS*]$ this is formalized by the following lemma:

Lemma 7: Given two RA $r_1 = \phi(s_1)$ and $r_2 = \phi(s_2)$, if p_1 is always valuated before p_2 in r_1 and p_2 is always valuated before p_3 in r_2, if s_1 and s_2 are compatible, then p_1 cannot be valuated for the $(i+1)^{th}$ time before p_1 is valuated for the i^{th} time in $r_1 \times_{RA} r_2$.

Proof: Follows from property (8) and lemma 5.

Rendezvous composition is succesfully implemented between processes using monitors and semaphors (see [15]), as well as with 3-way handshake protocols over a network (see [16]). This means we can compose RAs located at different sites across networks.

B. Correctness proof

In order to argue that we can distribute a synchronous system across a network we prove theorem 2 stated as follows:

Theorem 2: the compostion operator $\times_{RA}$ satisfies property (3), i.e.ϕ is a monomorphism between (STS*, $\times_{STS}$) and (RA, $\times_{RA}$)

Proof: Assume that the theorem does not holds. Then the following must hold (the proof for the other implication is simmetrical):

$$\begin{aligned}\exists s, s' \in STS*, \exists r, r' \in RA, \exists t \in \Gamma \quad & r = \phi(s) \wedge \\ & r' = \phi(s') \wedge \\ & r \times_{RA} r' \models t \wedge \\ & \neg(s \times_{STS^*} s' \models \\ & \quad \chi(t))\end{aligned}$$

where Γ is the set of traces of $r \times_{RA} r'$. This is to say that there exists at least a variable v and a natural i such that:

$$\chi(t)_i|v \neq \psi^v(\chi(t_i|P_I^{\psi^v} \cup P_S^{\psi^v})) \tag{10}$$

Select the smallest i for which (10) holds. Similarly fix one of the minimal variables v wrt $\prec$ of $s \times_{STS} s'$. Hence, by the minimality of v the following must hold:

$$\forall v' \prec v \,.\, \chi(t)_i|v' = \psi^{v'}(\chi(t)_i|P_i^{\psi^{v'}} \cup P_s^{\psi^{v'}}) = t_i^{v'} \tag{11}$$

Then (3) holds iff one $v' \in P_i^{\psi^{v'}} \cup P_s^{\psi^{v'}}$ has changed value after the i^{th} evaluation. This is possible, given (8), only if v' has been evaluated for the $i+1^{th}$ time before v has been evaluated for the i^{th} time. But this contradicts lemma (7), hence theorem 2 is proved. Q.E.D.

VII. CONCLUSION AND FUTURE WORK

We have addressed the problem of distributing large scale synchronous systems across a network. We defined a synchronous and asynchronous composition operator. The synchronous composition operator is SIMULINK-like. The asynchronous composition operator is CSP-style rendezvous [12]. We presented an algorithm to implement a synchronous program into an asynchronous one and we proved the implementation map preserves the synchronous semantics in the sense of [10]. The main result was that the implementation is a monomorphism with respect to the synchronous and asynchronous compositions. The monomorphism is our argument that a local change can be handled locally and that a subsystem can be re-used in different systems. The arguments of this paper were mathematical. We are currently working on software tools and libraries for the automatic distribution of SIMULINK programs. Our goal is to extend Simulink and Realtime Workshop to enable automatic distribution of systems across networks. In this paper we addressed the theoretical aspect of the problem, i.e. the preservation of the synchronous semantics and of the modularity after compilation. Other issues need to be addressed, such as data marshalling, remote naming and binding.

REFERENCES

[1] G. Berry, A. Benvenieste, *The synchronous approach to reactive and real- time systems*, Proceedings of the IEEE, 79(9):1270-1282, September 1991

[2] C. Andr, F. Boulanger, A. Girault, *Software entation of synchronous programs*, IEEE International Conference on Application of concurrency to System Design, June 2001

[3] C. Andr, M.A. Peraldi, *Effective implementation of ESTEREL programs*, 5th Euromicro workshop on real-time systems, June 1993.

[4] G. Berry, *The Constructive Semantics of Pure Esterel*, July 2, 1999

[5] N. Halbwachs, P. Caspi, P. Raymond and D. Pilaud, *The synchronous dataflow programming language Lustre*, Proceedings of the IEEE, vol. 79, nr. 9. September 1991.

[6] B. Houssais *The synchronous programming language SIGNAL, a tutorial*, IRISA, April 2002

[7] *Learning Simulink 5*, MathWorks edition, 2002

[8] A. Girault, *Automatic distribution of synchronous programs*, ERIM News, January 2003

[9] A. Girault, C. Menier, *Automatic production of Globally Asynchronous Locally Synchronous Systems*, EMSOFT 2002

[10] A. Benvenieste, B. Caillaud, P. Le Guernic, *Compositionality in dataflow synchronous languages: specification and distributed code generation*, Accademic Press, 2000

[11] N. Lynch, R. Segala, F. W. Vaandrager *Hybrid I/O automata*, Hybrid System III, LNCS 1066, Springer-Verlag, 1996, p.496-510

[12] C.A.R. Hoare, *Communicating sequential processes*, Prentice Hall, 2003

[13] Manna, Pnueli, *The temporal logic of reactive and concurrent systems*, Springer-Verlag 1992

[14] S. Edwards, *The specification and execution of Heterogeneous Synchronous Reactive Systems*, PhD thesis, University of California at Berkeley, 1997

[15] J. L. Hennessy, D.A. Patterson, D. Goldberg, *Computer Architecture: A quantitative approach* 3^{rd} edition, Morgan Kaefmann, 2002

[16] A. S. Tanenbaum, M. van Steen, *Distributed Systems, Principles and Paradigms*, Prentice Hall 2002