Distributing Synchronous Programs Using Bounded Queues*

Marco Zennaro
C³UV - Center for Collaborative Control of Unmanned Vehicles+
University of California at Berkeley
zennaro@path.berkeley.edu

Raja Sengupta
C³UV - Center for Collaborative Control of Unmanned Vehicles+
University of California at Berkeley
raja@path.berkeley.edu

ABSTRACT

This paper is about the modular compilation and distribution of a sub-class of Simulink programs [9] across networks using bounded FIFO queues. The problem is first addressed mathematically. Then, based on these formal results, a software library for the modular compilation and distribution of Simulink programs is given. The performance of the library is given. The value of synchronous programming for the next generation of traffic control is discussed. The adoption of these tools seems to be the natural candidate to address the needs of traffic engineers. As a case study we present an implementation in Simulink of a controller for coordinated traffic signals in an asymmetric peak hour traffic scenario and we evaluate its computational performance in a distributed environment.

Categories and Subject Descriptors: F.3.2 [Logics and meanings of programs]: Semantics of Programming languages

General Terms: Algorithms, Performance, Design, Languages, Theory

Keywords: Distributed synchronous programs, GALS, globally asynchronous locally synchronous architecture, Simulink.

1. INTRODUCTION

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],[3],[4]. The compiler takes care of translating the synchronous system into sequential code while preserving its semantic [4]. Synchronous programming languages like ESTEREL [5]-[6], LUSTRE [7], SIGNAL [8], or Simulink [9] 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, due to the asynchronous nature of the communication medium. 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. [10]-[11] propose algorithms to distribute synchronous 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 [12].

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 recompilation 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.

[12] 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 [13] we proposed such an algorithm based on CSP style randezvous [14]. In this paper we present an enhanced approach for the distribution.

Our approach is most similar to [15] and [16]. In [15] a blocking scheme is used to distribute discrete event systems. In the discrete event system setting particular attention has to be paid to avoid deadlock and livelock, while we prove this is not necessary for the class of problem we address. In [16] microcircuit components are composed together under the assumption that they are stallable, and the communication between components is modeled using fix sized FIFO queues.

Synchronous programs are here modelled using a finitary version of the Synchronous Transition System [17], modified to resemble Simulink. The asynchronous formalism is similar to the I/O automata of [18]. Synchronous and asynchronous composition operators are then defined. The synchronous composition operator is Simulink-like. The asynchronous composition operator is similar to the one used in Kahn Process Networks [19], [20], [21], but we assume the communication queues to have bounded size so that it can be implemented over reliable FIFO channels.

An implementation algorithm to map synchronous programs to asynchronous ones is then given and it is proven that the implementation map preserves the synchronous semantics in the sense of [12]. The main result is that the

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

+CCIT - 2105 Bancroft Way, Suite 300 Berkeley, CA 94720-3830

EMSOFT'05, September 19–22, 2005, Jersey City, New Jersey, USA.

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 theoretical results are then transformed into software. The architecture of the BSDP library and its performances are presented. The results in this paper apply only to Simulink programs without causal loops (see section 2) used with discrete fixed-rate solver. In [22] Simulink program are distributed over TTA networks. The BSDP library can be used on any kind of network: our compilation targets execution in a network of sequential machines communicating over any reliable FIFO channels with bounded memory. This execution model fits the GALS architecture. The class of Simulink programs we consider lie within the endochronous programs [12].

Our compilation process does no global scheduling computation. Thus if a block is changed, only the block itself needs to be re-compiled. On the other hand, our methods only preserve the synchronous semantic in the sense of the logical order of computation. It does not try to meet any real-time deadlines (as done, for example, in [23]).

The paper is organized as follows. Section 2 introduces a formalism for synchronous systems and section 3 one for asynchronous systems. Section 4 formulates the research problem mathematically. Section 5 presents the map from synchronous to asynchronous systems, compares it with the one used by Simulink, and proves the map preserves the synchronous semantic. Section 6 presents the main theorem supporting the distribution of Simulink programs. Section 7 presents the BSDP library architecture while 8 presents its performances. Section 9 presents a coordinated traffic control system developed using the tools introduced in this paper.

2. SYNCHRONOUS SYSTEMS

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

2.1 STS and FSTS

The Synchronous Transition System formalism, was introduced by Manna and Pnueli in [17]. STS describes a system as a tuple of typed state variables and transitions. Its behaviour is described through traces, i.e. an infinite sequence of states where a state is a valuation of all the variables of the system.

In this paper the Finitary STS (FSTS) formalism is used. The FSTS is chosen to relate to Simulink. A system is described in term of input and output ports, and internal state variables. The evolution of the system is captured by a set of functions used to compute the output and update the state.

Definition 1. A **Finitary Synchronous Transition System** (FSTS) is a tuple $(S, I, O, \sigma_0, \psi_O, \psi_S, \prec)$ where:

2.a. S is the finite set of state variables of the system;

2.b. I is the finite set of input ports of the system. I and S are required to be disjoint.

2.c. O is the finite set of output ports of the system. O and S are required to be disjoint. O and I are not necessarilly disjoint (this is needed for feedback as illustrated in the second example in section 2.2).

2.d. $\sigma_0(S)$ is the initial valuation of the state variables. $\sigma_0(s)$ denotes the initial value of the variable $s \in S$.

2.e. Ψ_O is a set of computable functions indexed by the output ports, used to compute the system outputs. ψ_o denotes the function indexed by the output port o.

2.f. Ψ_S is a set of computable functions indexed by the state variables, used to compute the next system state. ψ_s denotes the function indexed by the state variable s.

2.g. $\prec$ is an acyclic partial order over $I \cup O$ expressing the causality relation between input and output ports. Assume for example that the output o_i is the sum of the two imputs i_1 and i_2. Then o_i depends upon i_1 and i_2, written $i_1 \prec o_i$ and $i_2 \prec o_i$. If P is a set of ports then $\forall p \in P \,.\, p \prec p'$ is written as $P \prec p'$.
The concepts $\prec$ and I_p are linked: $\prec$ is defined as follows:

$$(\alpha, \beta) \in \prec \Leftrightarrow (\exists \psi_p \in \Psi_O \,.\, \alpha \in I_p \wedge \beta = p) \qquad (1)$$

In the following sections $P = S \cup O \cup I$ and $\Psi = \Psi_O \cup \Psi_S$ and suscripts are used when more than one FSTS are used (e.g. $\Psi_O^{s_1}$ refers to the set of output port functions of the FSTS s_1).

A Simulink block can be described by its I/O ports, state variables and the function used to update them. Later we capture Simulink using FSTS to make it work in a distributed computing environment. Some examples are given in the next section (2.2).

2.2 FSTS examples

Consider the simple Simulink system in figure (1.a). It is composed of a single gain block. It reads from the input port i_1 and outputs its value multiplied by two on the port o_1.

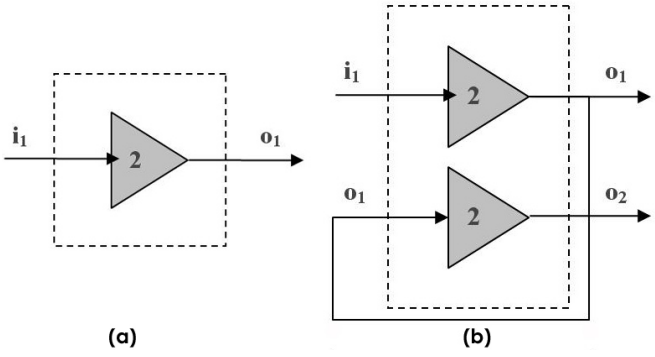

Figure 1: Two examples of Simulink systems

This system can be described as an FSTS $(S,I,O,\sigma_0(S),\Psi_O, \Psi_S,\prec)$ where $S = \emptyset$, $I = \{i_1\}$, $O = \{o_1\}$, $\sigma_0(S) = \emptyset$, $\Psi_S = \emptyset$, $\Psi_O = \{\psi_{o_1} \stackrel{def}{=} 2 * i_1\}$, $\prec = \{(i_1, o_1)\}$.

Notice that for the example in figure (1.a) $I \cap O = \emptyset$. In the example in figure (1.b), $I \cap O \neq \emptyset$. It is a block that

accepts two inputs i_1 and i_2 and has two outputs o_1 and o_2. o_1 and o_2 are twice i_1 and i_2 respectively.

As a result o_2 is four times i_1. This system can be described as the FSTS $(S, I, O, \sigma_0(S), \Psi_O, \Psi_S, \prec)$, where $S = \emptyset$, $I = \{i_1, o_1\}$, $O = \{o_1, o_2\}$, $\sigma_0(S) = \emptyset$, $\Psi_S = \emptyset$, $\Psi_O = \{\psi_{o_1} \stackrel{def}{=} 2 * i_1, \psi_{o_2} \stackrel{def}{=} 2 * o_1\}$, $\prec = \{(i_1, o_1), (o_1, o_2)\}$.

2.3 FSTS semantics

The semantic is given in terms of traces. Given a set of variables V, $\sigma(V)$ denotes a valuation of them and $\Lambda(V)$ the set of possible value assumed by the variables in V.

As for STS systems a trace is defined as follows:

Definition 2. A **trace** is an infinite sequence of valuations of $S \cup I \cup O$. The i^{th} vector of valuations in a trace t is denoted t_i, where $t_i \in \Lambda(S \cup I \cup O)$.

$t|P$ denotes the projection of the trace t over the set of ports and/or variables P.

Definition 3. **Tuple satisfaction**: given a trace t, the tuple t_i satisfies the system s, denoted $s \models t_i$, if the following holds:

$$\begin{aligned} s \models t_i \quad \Leftrightarrow \quad & (i = 0 \Rightarrow \forall s \in S \,.\, t_0|s = \sigma_0(s)) \wedge \\ & \forall p \in O \,.\, t_i|p = \psi_p(t_i|(I_p \cup S_p)) \wedge \\ & \forall s \in S \,.\, t_{i+1}|s = \psi_s(t_i|(I_s \cup S_p)) \end{aligned}$$

Where the semantic of function application is assumed to have no side effect.

Definition 4. **Trace satisfaction**: An FSTS system s admits a trace t (or equivalently the trace t satisfies the system s), written $s \models t$, as follows:

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

where $\mathbb{N}$ denotes the set of natural numbers including 0.

If $\prec$ is acyclic each t_i and a valuation of the inputs at time $i+1$ dictates an unique t_{i+1}. On the contrary, if $\prec$ has a cycle, there may be zero or multiple possibilities for t_{i+1}. Some authors have assumed out cycles [11], while others have looked for a fixed-point solution [24]. In this paper we follow the first approach. Thus every FSTS is *input deterministic*, i.e. given an input there is only one possible behaviour.

2.4 Compatible FSTS composition

In this section a composition operator for FSTS is defined. Once again, this is choosen to include Simulink. A complex system is composed of subsystems with interconnected inputs and outputs ports. Not all systems can be composed.

Definition 5. Two FSTS systems s_1=(S^{s_1},I^{s_1},O^{s_1}, $\sigma_0^{s_1}(S^{s_1})$, $\Psi_O^{s_1}$,$\Psi_S^{s_1}$,$\prec^{s_1}$) and s_2=(S^{s_2}, I^{s_2}, O^{s_2}, $\sigma_0^{s_2}(S^{s_2})$, $\Psi_O^{s_2}$, $\Psi_S^{s_2}$, $\prec^{s_2}$) are **compatible** if and only if:

6.a. $O^{s_1} \cap O^{s_2} = \emptyset$,
6.b. $S^{s_1} \cap S^{s_2} = \emptyset$,
6.c. $S^{s_1} \cap (O^{s_2} \cup I^{s_2}) = \emptyset$,
6.*d*. $S^{s_2} \cap (O^{s_1} \cup I^{s_1}) = \emptyset$,
6.*e*. $I^{s_1} \cap I^{s_2} = \emptyset$,
6.*f*. $\prec_a \cup \prec_b$ is acyclic.

The first condition ensures the two subsystems do not race to write the same output (this would introduce nondeterminism). The second, third and fourth conditions ensure that state variables are local and not shared between components. The fifth condition ensures that every input is received by a unique subsystem and that one output cannot be read by more than one inputs (this is not a limitation as it can be seen in the fourth example in 2.2). The last condition ensures the composed system does not have cyclic causal dependencies between variables.

Definition 6. The **composition** $s_1 \times_{FSTS} s_2$ = (S,I,O, $\sigma_0(S)$, Ψ_O,Ψ_S,$\prec$) of two compatible FSTS is defined as follows:

7.a. $I = (I^{s_1} \cup I^{s_2})$,
7.b. $P_O = (O^{s_1} \cup O^{s_2})$,
7.c. $P_S = (S^{s_1} \cup S^{s_2})$,
7.d. $\sigma_0(S) = (\sigma_O^{s_1}(S^{s_1}) \cup \sigma_O^{s_2}(S^{s_2}))$,
7.*e*. $\Psi_O = \Psi_O^{s_1} \cup \Psi_O^{s_2}$,
7.*f*. $\Psi_S = \Psi_S^{s_1} \cup \Psi_S^{s_2}$,
7.*g*. $\prec = (\prec_a \cup \prec_b)$.

In the following sections $\times_{FSTS}$ is denoted with $\times$ when it will not cause confusion.

Notice that $s_1 \times s_2$ is an FSTS because the compatibility hypothesis ensures there are no circular dependences between ports preserving input determinism. As defined, $\times_{FSTS}$ is a partial function over the FSTS set, i.e. it is defined only for compatible FSTS.

Some examples are given in the next section (2.5).

Next we state two simple propositions. The propositions merely assert our FSTS formalism has the usual properties of other formalisms for synchronous systems in the literature. A rigorous proof of them can be found in [25].

Proposition 2.1. *(FSTS, $\times_{FSTS}$) is a commutative monoid, with the identity element being the empty FSTS.*

Proposition 2.2. *Given two FSTS s_1 and s_2,*

$$s_1 \times_{FSTS} s_2 \models t \Leftrightarrow s_1 \models t|P^{s_1} \wedge s_2 \models t|P^{s_2}$$

2.5 FSTS composition examples

Consider the Simulink system in figure (2.a). The system is composed of two blocks similar to the one described in section 2.2. Both multiply the input but they do so by different factors;

The composed system is described as: $I = \{p_1, p_2\}$, $O = \{p_2, p_3\}$, $S = \emptyset$, $\sigma_0(S) = \emptyset$, $\Psi_S = \emptyset$, $\Psi_O = \{\psi_{p_2} \stackrel{def}{=} 2 * p_1, \psi_{p_3} \stackrel{def}{=} (3 * p_2)\}$, $\prec = \{(p_1, p_2), (p_2, p_3)\}$.

The composed system has the expected semantic. It multiplies the input by 6.

It may apear that the compatibility conditions as defined in (5.a) are too restrictive, ruling out systems where the output of a block is feeded to more than one subsystem. This is not the case as illustrated by the example in figure (2.b).

The system has three subsystems. Two of them are the gain blocks described in the previous examples. The third one is the *duplicate* block that is formally described as: $I = \{i_1\}$, $O = \{o_a, o_b\}$, $S = \emptyset$, $\sigma_0(S) = \emptyset$, $\Psi_S = \emptyset$, $\Psi_O = \{\psi_{o_a} \stackrel{def}{=} i_1, \psi_{o_b} \stackrel{def}{=} i_1\}$, $\prec = \{(i_1, o_a), (i_1, o_b)\}$.

The composition of the three block is described with the following FSTS: $I = \{i_1, o_a, o_b\}$, $O = \{o_a, o_b, o_1, o_2\}$, $S = \emptyset$, $\sigma_0(S) = \emptyset$, $\Psi_S = \emptyset$, $\Psi_O = \{\psi_{o_a} \stackrel{def}{=} i_1, \psi_{o_b} \stackrel{def}{=} i_1, \psi_{o_1} \stackrel{def}{=} 3 * o_a, \psi_{o_2} \stackrel{def}{=} 3 * o_b\}$, $\prec = \{(i_1, o_a), (i_1, o_b), (o_a, o_1), (o_b, o_2)\}$.

3. ASYNCHRONOUS SYSTEMS

There are many asynchronous system formalisms in the literature. One of them is the asynchronous version of STS,

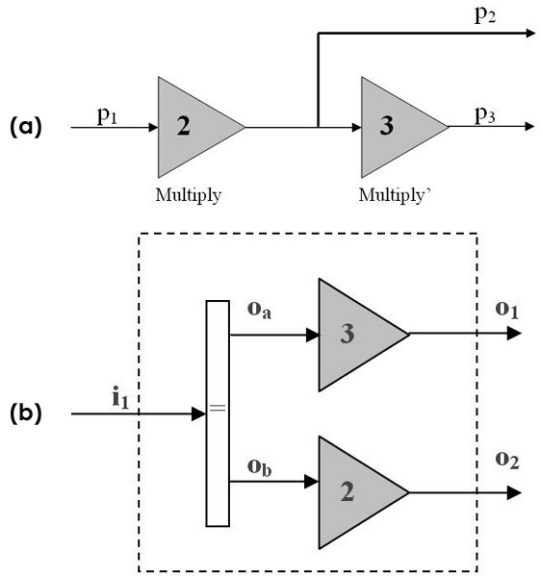

Figure 2: Simulink systems composed of multiple blocks

called the Asynchronous Transition System (ATS) model, introduced by Benvenieste in [3]. 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). The simplicity of the model makes it easy to handle it mathematically, but we seek a finitary formalism to be output of an algorithm.

Instead we use automata augmented with queue variables. We call them Reactive Automata (RA). A reactive automaton is a labeled finite automaton communicating through shared queues. It is a discrete version of the IO-automata described in [18] augmented with communication ports. $\mathbb{V}$ denotes the set of variables, $\mathbb{P}$ the set of ports and for any port p in $\mathbb{P}$, $\beta(p)$ is the bound (maximum capacity) of the queue p. Formally an RA is a tuple $(L, l_0, V, \sigma_0(V), P_I, P_O, T)$ where

- L is a finite set of locations of the automaton;
- l_0 is the initial location, $l_0 \in$ L;
- V is a finite set of variables read and written only by the RA;
- $\sigma_0(V)$ is the initial value of the state variables;
- P_I is a finite set of communication ports, considered as environmental queues read by this RA;
- P_O is a finite set of communication ports, considered as environmental queues, written by this 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 the values of the elements in V. A is defined by the following grammar:
  $A \rightarrow ?p(v)$ where $p \in P_I$ and $v \in V$
  $A \rightarrow !p(v)$ where $p \in P_O$ and $v \in V$
  $A \rightarrow v := f(V_1)$ where $v \in V$, $V_1 \subseteq V, f \in \mathbb{F}(V_1)$ is the set of functions with the standard syntax of a term in first order logic (see [26]), where the symbols occouring are either function symbols or variable symbols in V_1.

In the following sections P denotes the set $P_I \cup P_O$.

An example of an RA is given in figure (3) and is fomalized as the following RA:

$$(\{W, P, S\}, W, \{v_1, v_2\}, \{0, 0\}, \{input\}, \{output\}, \{(W, P, True, ?input(v_1), (P, S, True, v_2 := v_1 + 1), (S, W, True, !output(v_2)\})$$

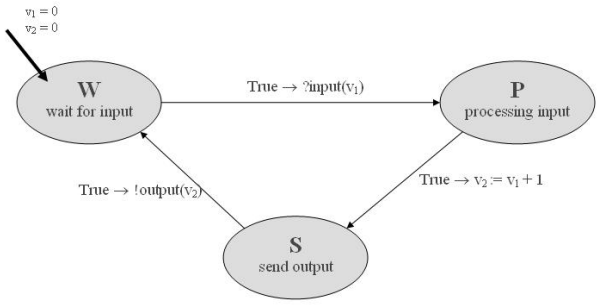

Figure 3: A simple reactive automaton

3.1 RA semantic

The semantic of an RA is in terms of runs and traces.

Definition 7. A **run** of a Reactive Automaton is an infinite sequence of (*location*, *variables valuation*, *transition*, *ports valuation*) tuples.

The actions are reads (denoted $?p(v)$), writes (denoted $!p(v)$), computations (denoted $v := f(V)$), and the silent action (denoted ϵ). The silent action is introduced to denote the reception or transmission of data in an input or output queue due to an action of the environment. A transition with an input action removes the element at the head of an input port and writes it to an internal state variable, while a transition with an output action adds the value of a variable to the tail of an output port.

Definition 8. A reactive automaton **trace** is a tuple, where each element of the tuple is an infinite sequence of valuations for a particular variable of the reactive automaton. The i^{th} valuation of a variable v in a trace t is denoted by $(t|v)_i$.

The following is a representation of the initial part of a run of the RA in figure (3) for the input port valuation {1} and the output port valuation $\emptyset$:

$$(W, (0, 0), T_{true} \rightarrow ?I_{nput}(v_1), (< 1 >, \emptyset)),$$
$$(P, (1, 0), T_{true} \rightarrow v_2 := v_{1+1};, (\emptyset, \emptyset)),$$
$$(S, (1, 2), T_{true} \rightarrow !O_{utput}(v_2), (\emptyset, \emptyset)), (W, (1, 2), -, (\emptyset, < 2 >)), \ldots$$

where W, P and S are the *wait for input*, *Process Input* and *Send Output* location respectivelly and the second element is a valuation of v_1 and v_2, and the third element is a valuation for the two ports *Input* and *Output*.

Thus mathematically a run is a sequence of tuples like the one above. The i^{th} tuple in a run r is denoted by r_i and its element are extracted using projection, for example $r_i|location$ denotes the location element of the tuple r_i.

Given a run, the **associated trace** can be computed by examining the update action on every variable of the RA, i.e. the i^{th} element of the sequence associated with the state variable v is given by the i^{th} update on that variable. A variable v can be updated in two possible ways: because of a read action $?p(v)$, or because of a computation action $v := f(V)$. Given a RA run $r =< r_0, r_1, r_2, ... >$, $(t|v)$ is computed extracting a sequence $< r_{k_0}, r_{k_1}, ...$ from r such that for all k_i $r_{k_i}|action$ is an update action for v and for all $j \neq k_i$ $r_j|action$ is not an update action for v. An update action for v is an input action on the form $?p(v)$ for any port p or an update action on the form $v := f(V)$, for any function f.

For the previous run, the associated trace is $< (0, 1, ...), (0, 2, ...) >$ where the first and the second sequences are the successive valuations of v_1 and v_2 respectively.

Definition 9. **Tuple satisfaction**: Given a reactive automaton run r, we say that the tuple r_i satisfies a RA w, denoted $w \models r_i$ iff the following holds:

$$
\begin{array}{l}
(i = 0 \Rightarrow \ (r_0|_{loc} = l_0 \wedge r_0|V = \sigma_0(V) \wedge \forall p \in P_0 \ r_0|p = \emptyset)) \wedge \\
(r_i|action = \epsilon \Rightarrow \forall v \in V \ . \ r_i|v = r_{i+1}|v \wedge \\
\qquad \forall p \in P_O \ . \ (r_i|p = r_{i+1}|p \vee r_{i+1}|p = tail(r_i|p)) \wedge \\
\qquad \forall p \in P_I \ . \ (r_i|p = tail(r_{i+1}|p)) \vee (r_i|p = r_{i+1}|p)) \vee \\
(\exists (s, s', (c, a)) \in T \ \Rightarrow \ r_i|_{location} = s \ \wedge r_{i+1}|_{location} = s' \wedge \\
\qquad c \models r_i|(V \cup P) \wedge r_{i+1}|(V \cup P) = act(a, r_i|(V \cup P)))
\end{array}
$$

Observe that the values of a port may change value without any input or output by the component, by its environment, simulating the reception of a message through that port, through an ϵ-transition. At the same time, by the definition of *act* in the next paragraph, input actions on empty input ports and output actions on full output ports are not defined. Hence input and output actions are blocking.

Assume for now that $P_I \cup P_O = \{p_1, .., p_m\}$ and that $V = \{v_1, .., v_n\}$. Then the function *act* is defined as follows:
$act(a, \sigma(p_1), .., \sigma(p_m), \sigma(v_1), .., \sigma(v_n)) =$

$$
\begin{cases}
(\sigma(p_1), .., \sigma(p_m), \\
\sigma(v_1), .., \sigma(v_{j-1}), \sigma(f)(\sigma(v_{i_1}), .., \sigma(v_{i_k})), \sigma(v_{j+1}), .., \sigma(v_n)) \\
\qquad \text{if } a = \text{“}v_j := f(v_{i_1}, .., v_{i_k})\text{”} \\
(\sigma(p_1), .., \sigma(p_{j-1}), push(\sigma(v_i), \sigma(p_j)), \sigma(p_{j+1}), .., \sigma(p_m), \\
\sigma(v_1), .., \sigma(v_n)) \\
\qquad \text{if } a = \text{“}!p_j(v_i)\text{”} \ \wedge \neg full(\sigma(p_j)) \\
(\sigma(p_1), .., \sigma(p_{j-1}), tail(\sigma(p_j)), \sigma(p_{j+1}), .., \sigma(p_m), \\
\sigma(v_1), .., \sigma(v_{i-1}), head(\sigma(p_j)), \sigma(v_{i+1}), .., \sigma(v_n)) \\
\qquad \text{if } a = \text{“}?p_j(v_i)\text{”} \ \wedge \neg empty(\sigma(p_j))
\end{cases}
$$

where $\sigma(.)$ denotes the variable and port valuation. The function *full*, *empty*, *head*, *tail* and *push* are the standard operations over bounded size queues. Assume the semantic of function application to be the same used in the case of FSTS. In particular, a function evaluation has no side effects.

Definition 10. **Run satisfaction**: A run r satisfies a reactive automaton w, denoted $w \models r$ iff:

$$\forall i \in \mathbb{N} \ w \models r_i$$

Definition 11. **Trace satisfaction**: 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.

We now define a composition operator $\times_{RA}$ for reactive automata.

Definition 12. Given two reactive automata (L^1, l_0^1, V^1, $\sigma_0^1(V^1)$, P_I^1,P_O^1, T^1) and (L^2, l_0^2, V^2, $\sigma_0^2(V^2)$, P_I^2, P_O^2 T^2) they are **compatible** if the following condition hold:

$$V^1 \cap V^2 = \emptyset \ \ \wedge \ \ P_O^1 \cap P_O^2 = \emptyset \ \ \wedge \ \ P_I^1 \cap P_I^2 = \emptyset.$$

The first conjunct requires the variables of each RA to be local. The last two say that two distinct automata cannot write the same port or read the same port.

Definition 13. **Reactive automaton composition**: Given two compatible reactive automata $w_1 = (L^1, l_0^1, V^1, \sigma_0^1(V^1), P_I^1, P_O^1, T^1)$ and $w_2 = (L^2, l_0^2, V^2, \sigma_0^2(V^2), P_I^2, P_O^2 \ T^2)$ Their composition $w_1 \times_{RA} w_2$ is defined as the automaton (L, l_0, V, $\sigma_0(V)$, P, T) where:

1. $L = \bigcup^{l_1 \in L^1, l_2 \in L^2} \{\{(w_1, l_1), (w_2, l_2)\}\}$
2. $l_0 = \{(w_1, l_0^1), (w_2, l_0^2)\}$
3. $V = V^1 \cup V^2$
4. $\sigma_0(V) = \sigma_0(V)^1 \cup \sigma_0(V)^2$
5. $P_I = (P_I^1 \cup P_I^2)$
6. $P_O = (P_O^1 \cup P_O^2)$
7. $T = \{(s, d, c, a) | ((s|L_1, d|L_1, c, a) \in T^1) \wedge (s|L_2 = d|L_2)) \vee ((s|L_2, d|L_2, c, a) \in T^2) \wedge (s|L_1 = d|L_1))\}$
   This is an interleaving of the executions of the two original automata.

Lemma 3.1. *(RA, $\times_{RA}$) is a commutative monoid, with the identity element being the empty RA.*

A proof of the lemma is given in [25].

$\prod_{w \in W} w$ denotes an n-ary composition of RA's. Lemma (3.1) shows this is well-defined as the usual extension of the binary operator $\times_{RA}$.

Definition 14. Given a run w of the automaton $\prod_{w \in W} w$, the projection of the product to one of the factors $w \in W$ is formally defined as follows:

$$
\begin{array}{l}
\forall i \in \mathbb{N} \ . \ (r|_w)_i|_{location} = l \wedge (w, l) \in (r_i|_{location}) \wedge \\
\quad \forall v \in V^w \ . \ (r|_w)_i|_v = (r_i|_v) \wedge \\
\quad (r_i|_{transition}) \in w \Rightarrow (r|_w)_i|_{transition} = (r_i|_{transition}) \wedge \\
\quad (r_i|_{transition}) \notin w \Rightarrow (r|_w)_i|_{transition} = \epsilon \wedge \\
\quad \forall p \in (P_O^w \cup P_I^w) \ . \ (r|_w)_i|_p = (r_i|_p)
\end{array}
$$

Every tuple r_i of the run of the product is projected to the variables and locations of w and the tuple with transition not belonging to $w|T$ are replaced with a silent transition.

Lemma 3.2. *Given two compatible reactive automata w_1 and w_2 and given a run r of their composition, the following holds:*

$$(w_1 \times w_2 \models r) \Rightarrow (w_1 \models r|w_1 \wedge w_2 \models r|w_2)$$

A proof of the lemma is given in [25].

RA can be easily compiled to run on a sequential machine. A product of reactive automata could be compiled in a few ways. The composition can be carried out generating a third automaton, or the two original automata can be run in parallel as long as the following hyphothesis (embedded in our definition of satisfaction) holds:

Hyphothesis 3.3. *The communication queues are FIFO queues, the values are not lost and their order is maintained.*

In the second approach the composition can be implemented within a single machine between processes using monitors and semaphors (see [27]), as well as with 3-way handshakes protocols over a network (see [28]). This means we can compose RAs located at different sites across networks. In section 7 we will explore an approach that takes full advantage of the distribution of the code (maximising pipeline gain).

4. PROBLEM STATEMENT

Given the definition of FSTS and RA in the previous sections, we can now formally define our problem. Figure 3 illustrates the research program. First we need to find a way to associate RA and FSTS traces, that is to say we need a trace map $\chi : \mathbb{T}_{RA} \rightarrow \mathbb{T}_{FSTS}$ where $\mathbb{T}_{RA}$ and $\mathbb{T}_{STS}$ are the set of traces of STS and RA respectively. In [12] the following definition of χ is given:

Definition 15. $t' = \chi(t) \Leftrightarrow \forall i \in \mathbb{N}\ \forall v \in V\ .\ (t|_v)_i = t'_i|_v$

We need to find a way to implement FSTS as RA while preserving the synchronous semantic, that is to say we need to find an implementation map $\phi : \mathbb{F}\text{STS} \rightarrow \mathbb{R}\text{A}$ such that the following holds:

$$\forall w \in R_A\ \forall s \in F_{STS}\ .\ w = \phi(s) \Rightarrow (r \models t \Leftrightarrow s \models \chi(t)) \quad (2)$$

If this holds then ϕ maps a synchronous system into an asynchronous system while preserving the synchronous semantic. It has been proved in [12] that for the set of *endochronous* programs such a ϕ exists. In section 5 we define a ϕ for the class of FSTS.

So far we have just obtained what a Simulink compiler does, or what is done in [4]. Given such maps we can now formulate our problem (like [12]) as follows: we seek a composition operator $\times_{RA}$ such that, for any two FSTS s_1 and s_2 and RA w_1 and w_2, the following holds:

$$\begin{array}{c} w_1 = \phi(s_1)\ \wedge w_2 = \phi(s_2)\ \Rightarrow \\ (w_1 \times_{RA} w_2 \models t \Leftrightarrow s_1 \times_{STS} s_2 \models \chi(t)) \end{array} \quad (3)$$

If this holds and if the composition operator $\times_{RA}$ can be implemented across a network then 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 [12] that when the pair (s_1, s_2) is *isochronous* than such an operator exists. In section 6 we prove that property (3) holds if the two synchronous system are *compatible* (as defined in section 2). Thus we claim ϕ is a monomorphism between ($\mathbb{F}$STS, $\times_{FSTS}$) and ($\mathbb{R}$A, $\times_{RA}$).

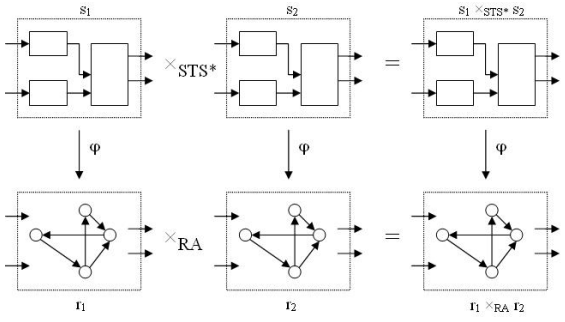

Figure 4: A graphical representation of property (3)

5. IMPLEMENTATION OF FSTS SYSTEMS

In this section ϕ, a mapping of FSTSs into RAs is given. It is then proven that the ϕ satisfies(2).

ϕ is defined by the following algorithm:

Algorithm Φ
Inputs: an FSTS $s=(S, I, O, \sigma_0(S), \psi_O, \psi_S, \prec)$
Outputs: An RA $r=(L, l_0, V, \sigma'_0(V), P_I, P_O, T)$ *that implements the input system*

$P_I := \{p_j | j \in I \backslash O\}$
$P_O := \{p_j | j \in O \backslash I\}$
$V = I \cup O \cup S$
$\forall i \in (I \cup O)\ .\ \sigma'_0(i)| = 0$
$\forall j \in S\ .\ \sigma'_0(j) = \sigma_0(j)$
$l_0 := l_{root}$
$(N, E) := CG(\prec |(I \cup O), (I \cup O), root, leaf)$
For all $n \in N$ add l_n in L
For all $(n, n', j) \in E$ do
  if $j \in (I \backslash O)$ then do
    add $(l_n, l_{n'}, (true, ?p_j(j)))$ to T
  od
  if $j \in (O \backslash I)$ then do
    add $l_{n,j}$ in L
    add $(l_n, l_{n,j}, (true, j := \psi_j(V|P_j)))$ to T
    add $(l_{n,j}, l_{n'}, (true, !p_j(j)))$ to T
  od
  if $j \in (O \cap I)$ then do
    add $(l_n, l_{n'}, (true, j := \psi_j(V|P_j)))$ to T
  od
od
Let $<$ be any linearization of $\prec|_S$
$(N, E) := CG(<), (S), leaf, root)$
For all $n \in N$ add l_n in L
For all $(n, n', j) \in E$ do
  add $(l_n, l_{n'}, (true, j := \psi_j(V|P_j))$ to T
od

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

$Nodes := \{root, leaf\}$
$Edges := \emptyset$
% max-int is a global variable that holds the highest
% integer used to label a node
$counter := max - int + 1$
$\forall$ linearization $w = (w_1, w_2, ..., w_m)$ of $\prec$ in P do
  $pointer = root$
  For all $i \in [1, m]$ do
    if $(pointer, n, w_i) \in Edges$ do $pointer = n$
      else do
        add $n_{counter}$ to $Nodes$
        add $(pointer, n_{counter}, w_i)$ to $Edges$
        $pointer := n_{counter}$
        $counter++$
      od
  od
od
Replace the sinks in $Nodes$ and $Edges$ with $leaf$

The algorithm is guaranted to terminate for every FSTS. All the for loops terminate in finitely many steps because the set of variables and ports of an FSTS is finite. If $\prec$ is

not acyclic then the algorithm cannot be applied because $\prec$ would not be linearizable.

The first theorem stated below asserts algorithm ϕ constructs an RA implementing of an FSTS while preserving its semantics in the sense of χ.

Theorem 5.1. *Algorithm ϕ satisfies property (2), i.e.*

$$\forall w \in RA\, \forall s \in FSTS\ .\ w = \phi(s) \Rightarrow (r \models t \Leftrightarrow s \models \chi(t))$$

A proof of the theorem can be found in [25].

A Simulink program goes through the following phases: it starts in the initialization phase computing sample times and parameters, determining the block execution order and allocating memory. Then the loop phase starts, where the following steps are repeated: read the input (input step), compute the output and propagate it (output step) and update the state (state step). Last in the termination phase the memory is released.

In Simulink programs without causal loops, the order of computation produced in the initialization step is computed through a linearization of the causality relation between inputs and outputs.

The algorithm used by Simulink (Real-Time workshop) for the simulation (implementation) of a system is hence different from the one given in the previous section. For single rate systems with no causal loops the main difference is that an FSTS is not mapped into a RA able to receive its inputs in all the possible orders, but only in a particular order. The subroutine CG is no longer necessary and line 7 is replaced with a routine that constructs a single path graph. Alternativelly we can just pass to the CG routine a linearization of $\prec$ instead of $\prec$. In the next sections ϕ_{sim} denotes the algorithm with this modifications.

All the claims and proof of the previous section will hold for ϕ_{sim} as well. However in the next sections it is showed that ϕ can be distributed with fewer assumption than ϕ_{sim}.

6. DISTRIBUTION OF FSTS SYSTEMS

We have seen in the previous section that there is a map ϕ between FSTS and RA satisfying property (2). We have claimed in section 3 that $\times_{RA}$ can be implemented across comunicating machines. Hence, we argue that we can distribute a Simulink-like synchronous system across a network with the following theorem (a proof for it is given in [25]).

Theorem 6.1. *The compostion operator $\times_{RA}$ satisfies property (3), i.e. for any two compatible FSTS $s = (S^s, I^s, O^s, I^s_O, \Psi^s_O, \Psi^s_S, \prec^s)$ and $s' = (S^{s'}, I^{s'}, O^{s'}, I^{s'}_O, \Psi^{s'}_O, \Psi^{s'}_S, \prec^{s'})$ the following holds:*

$$\forall t \in \Gamma\ .\ \phi(s) \times_{RA} \phi(s') \models t \Leftrightarrow s \times_{STS} s' \models \chi(t)$$

As noted in section 5 the implementation algorithm used by Matlab Simulink / RealTime Workshop differs from ϕ proposed for FSTS in the sense that it fixes the order in which the input are received and the outputs are computed and propagated to the other subsystems.

Theorem 6.1 do not extend in the general case for ϕ_{sim}. It suffices to consider the FSTS in figure (5) (taken from [12]).

It is easy to see that s_0 cannot be compiled through ϕ_{sim} without deadlocking if composed with s_1 or s_2. If it is compiled to accept i_1 before i_2 then it will block if composed with s_2. If compiled to accept i_2 before i_1 it will deadlock when composed with s_1. In reality a Simulink systems

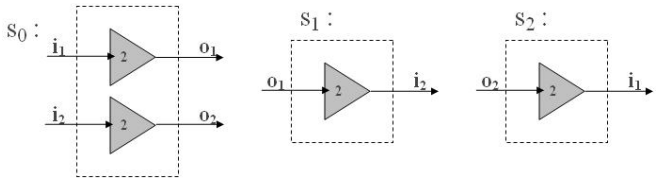

Figure 5: Three FSTS systems

reads all the inputs before computing any of the outputs. This means that s_0 will deadlock with both s_1 and s_2.

This shows that as long as the Matlab Simulink interpreter / Realtime Workshop compiler is used, synchronous systems cannot be distributed in the general case. However it can be done in the following particular case:

Theorem 6.2. *Given an FSTS $s' = \prod_{s \in S} s$, if $\prec^{s'}$ projected to the ports of each subsystem s is a total order (i.e. the external outputs depends on all the external inputs), then for any two compatible FSTS $s = (S^s, I^s, O^s, I^s_O, \Psi^s_O, \Psi^s_S, \prec^s)$ and $s' = (S^{s'}, I^{s'}, O^{s'}, I^{s'}_O, \Psi^{s'}_O, \Psi^{s'}_S, \prec^{s'})$ the following holds:*

$$\forall t \in \Gamma\ .\ \phi_{sim}(s) \times_{RA} \phi_{sim}(s') \models t \Leftrightarrow s \times_{STS} s' \models \chi(t)$$

Proof: Since $\prec^{s'}$ projected over the subsystems is a total order the output of CG is a single path graph with root l_{root} and sink l_{leaf}. As a result ϕ and ϕ_{sim} produce the same output. The theorem follows.

7. BDSP ARCHITECTURE

In this section the software architecture for the distribution of Simulink programs (see figure (6)) is described. We call this architecture Berkeley Distributed Simulink Program (BDSP) library.

An initial version of the BDSP library has been implemented using a simple rendezvous scheme. The first version was developed as a proof of concept, a second version, utilising bounded queues as described in this section is currently under development.

The current implementation relies on the Simulink interpreter. Because of it the systems are distributed as follows: first the original Simulink model is decomposed into atomic blocks. Then all the broken connections are replaced with $external - linkboxes$ (i.e. S-function boxes we provide). These boxes hide the complexity of the distribution to the user.

Input and Output external-link boxes structure: the structure of an Input external-link box and of an Output external-link box are the same but for the ports. While the input box has a single input and no outputs the output box should have one output and no inputs. The boxes have three parameters: the IP/port pair for the sender, the IP/port pair for the receiver and a name that is going to be used to resolve for the first two parameters. The box uses two TCP sockets to communicate with the queue manager. One socket is used to receive messages from the queue manager and the second is used to send messages to it.

Queue Manager structure: the structure of the queue manager is shown in the right side of figure (6). It consists of many queues, one for every input or output port of the block. It has a couple of TCP sockets to comunicate with the S-function boxes on the machine and a list of UDP

sockets to communicate with the the other queue managers. Every queue is associated with two flags (the *datarequested* and *queuefull*) and a counter.

External-link box to queue manager interface: The life cycle of an external-link box is the same of any Simulink box (described in section 5). In the initialization phase the box sends a packet to the queue manager to reserve a queue and pass the IP/port address to the other end of the pipe. If it is an input block it requests its input from the queue manager in the Input Read phase. If the queue is empty it blocks until something is available. The flag *datarequested* is switched on if the queue is empty. If it is not empty the data is removed from the queue and sent to the box. If it is an output block, in the Ouput Phase the output is sent to the Queue manager. If the queue is not full an ack is sent back to the output box. The box is blocked until the ack is received. If the queue is full and the box is trying to send, the flag $Full$ is switched on. When the queue is empty and the flag $Full$ is on an ack is sent to the Output box.

Queue manager to queue manager interface: the communication protocol between queue managers needs to be reliable and to preserve message order. A possible candidate is TCP, or a UDP with a acknowledgment-timeout protocol implemented on top. When an output queue is not empty the queue manager will try to send the message as soon as possible. It removes the message from the queue only when the ack is received. When it receives a message it will put it on the right queue. If the queue is full it will drop the packet (the message will not be lost, just retransmitted later).

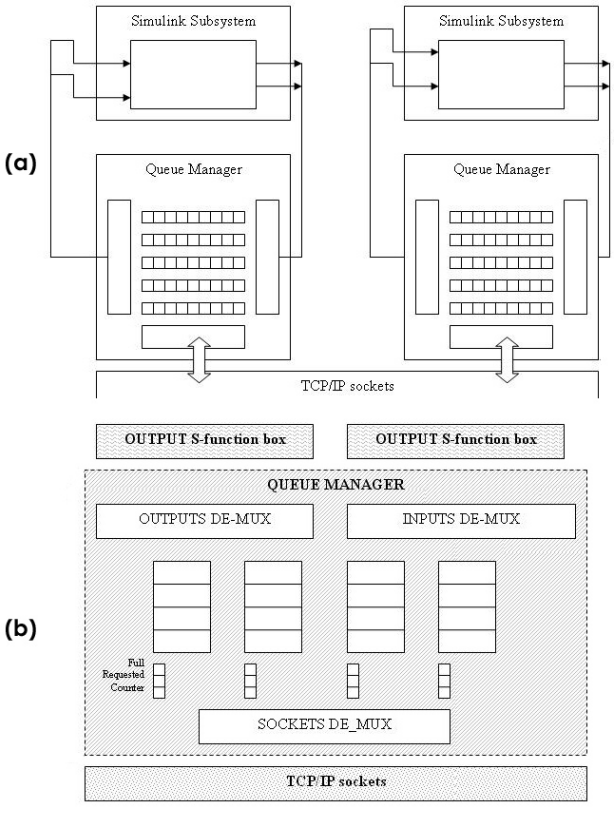

Figure 6: BDSP architecture

8. PERFORMANCE ANALYSIS

Code distribution may lead to a system speed-up through concurrency, but it has also a cost overhead associated with the rendezvous communication protocol. In this section this overhead is estimated for the first implementation of the BDSP library as described in section 7.

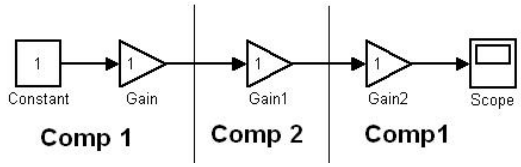

Figure 7: The model used to estimate the overhead

We decompose the system in figure (7) into three subsystems running on two separate Pentium 4 850 Mhz, 512 Mb ram machines. The source and the sink gain are located on the same machine, while the middle gain is run on a second one. A timestamp is recorded by the external-link boxes at the beginning and at the end of each time step. Since the source and sink gain are on the same machine, i.e. they are running according to the same clock, the time stamps can be compared to get a conservative estimate of the overhead due to the rendezvous protocol. The measured overhead is conservative because it includes the middle gain computation time and the two Simulink processes on the first processor are competing on the first computer. The computers are connected through a shared 802.11b wireless ethernet.

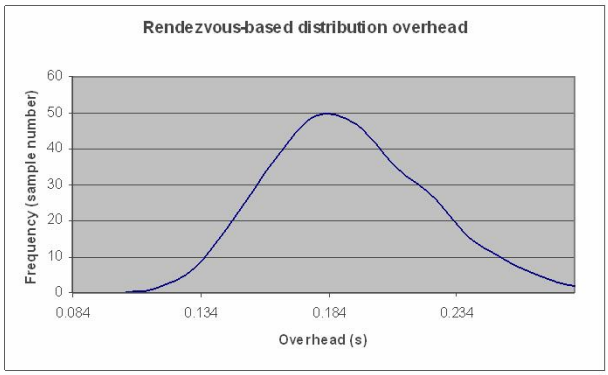

Figure 8: A conservative estimate of the distribution overhead

The results are plotted in figure (8). The overhead average is smaller that 0.2 seconds and the standard deviation is close to 30 ms. This result is promising considering that we are currently using the Simulink interpreter and not the Real-time workshop compiler.

9. TRAFFIC SIGNAL CONTROL APPLICATION

We are working to introduce synchronous programming techniques into traffic signal control. As they are growing rapidly in complexity we see an excellent opportunity for synchronous programming tools as a way to greatly simplify software development for these large-scale systems.

In order to maximize the flow and minimize the average waiting time, the cycle length (defined as the time needed to go through all the phases) and the interval splitting (defined as the ratio of the green time for the two directions), need to be properly set.

The early traffic signal controllers were non-programmable devices. However, with time these devices have reached a high level of sophistication. An example of such a device is the 2070 controller, used widely in California, which supports pre-timed, semi-actuated and fully actuated operation rules and supports a wide set of sensors. These devices support many pre-defined rules that can be adjusted on the field or remotely (in the case of the 2070 the remote setting protocol is fixed by the National Transportation Communications for ITS protocol set of standards).

At the same time, signal control systems are growing spatially. The first dynamically adjustable lights were driven by traffic measurement sensors located next to them. They were isolated. Next it became possible to coordinate all the lights along an arterial to have the lights turn green in succession. Modern systems like [29] seek to coordinate entire downtown urban grids. The entire grid is operated on a common cycle time adjusted on the timescale of tens of minutes as demand changes. Controllers like 2070 are only partially programmable and the programming interfaces are low-level. Furthermore, embedded computing and the wireless revolution are being brought together by the US governments Vehicle Infrastructure Initiative (VII) [30]. It is envisaged that every roadside cabinet will have a general purpose computer with wired or wireless backhaul. A large-signal control system could be developed in high-level tools like Simulink and compiled to suit the hardware architecture at hand. The entire system may compute in a traffic management center with low-level commands going out to the field, or be distributed to compute entirely in field cabinets.

As a first step we have used SIMULINK to model a major arterial road intersected by 4 minor low traffic streets. Consider a peak hour asymmetric scenario, where almost all the traffic flow is in one direction on the major street. The flow is maximized by coordinating the traffic lights to create green waves: a car that just got the right-of-way at the first intersection will get a green at all the intersections (see [31] and [32]). This is done by synchronizing the controllers, fixing the cycle length across controllers, and offsetting the begining of each cycle by a statically determined $d*v$, where d is the distance between the two intersections and v is the target traffic speed.

If the intersection has inductive loops the vehicle speed can be directly estimated. This value can be passed through a simple filter to make the system resistant to insignificant minor speed fluctuations, while adjusting to significant and permanent changes (due for example to congestion, road construction or minor accident).

This actuated scheme is implemented by the Simulink model in figure (9). The average speed in response to the traffic light has been computed using traffic flow theory as described in [35]. The sensor input is passed through a simple filter to make the system resistant to insignificant minor speed fluctuations, while adjusting to significant and permanent changes.

We have run a simulation of the system where an accident occurs between the first and the second intersection during the 100^{th} cycles and is cleared out during the 180^{th}, and a minor one happens between the third and the fourth intersection during the 150^{th} cycle and it is cleared out during the 200^{th}.

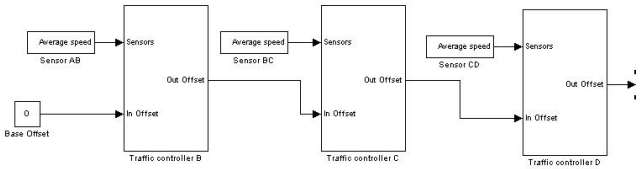

Figure 9: Traffic Controller Simulink Model

The offsets computed by the last three intersection (the offset for the first one is always 0) computed using the model in figure (9) are plotted in figure (10).

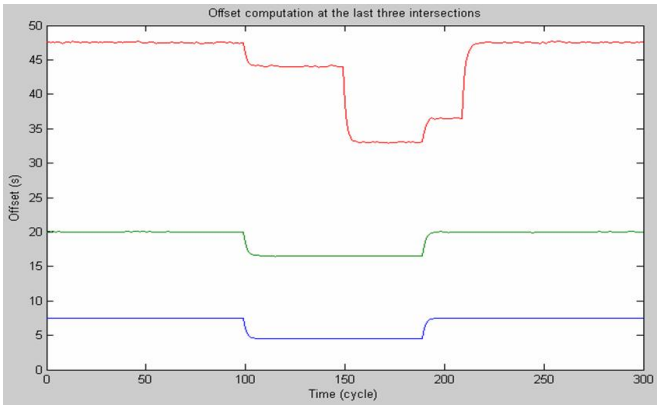

Figure 10: The offset as computed by the model described in figure (9)

This model can be compiled and run centrally at the traffic management center, making the traffic signal operating, quoting [33], "exactly the way the designer thinks it should be controlled". Moreover, as shown in sections 6, the model can be compiled into distributed code, with the same behaviour, that can be run by the controllers without any need of external coordination.

The test to evaluate the performances of the system has been carried over the same hardware used in the previous section. In this case the performances have been measured as the total computation time needed to carry a step (i.e. from the end of the previous cycle to the end of the computation of all the offsets). The computation time is on average 0.3 s (the standard deviation is 6 ms). We expect this result to improve when moving from Simulink interpretation to direct execution of the code as generated by Real-Time workshop. Even interpreting the code though, the system largely met the time constraints of the application as described in [33].

10. CONCLUSION

The problem of distributing large scale synchronous systems across a network has been addressed. We defined a synchronous and asynchronous composition operator. The synchronous composition operator is Simulink-like. The asynchronous composition operator is similar to the one used in Kahn process networks. We presented an algorithm to im-

plement a synchronous program into an asynchronous one and we proved the implementation map preserves the synchronous semantics in the sense of [12]. 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. We have presented a software architecture consistent with our mathematics and studied its performanmces. We have motivated the development of synchronous programming tools for traffic signal control.

11. REFERENCES

[1] E. A. Lee, *Concurrent Models of Computation for Embedded Software*, Technical Memorandum UCB/ERL M05/2, University of California, Berkeley, 2005

[2] E. A. Lee and Stephen Neuendorffer, *Concurrent Models of Computation for Embedded Software*, Technical Memorandum UCB/ERL M04/26, University of California, Berkeley, 2004.

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

[4] C. Andre', F. Boulanger, A. Girault, *Software implemenentation of synchronous programs*, IEEE International Conference on Application of concurrency to System Design, June 2001

[5] G. Berry, *The Foundations of Esterel*, Proof, Language and Interaction: Essays in Honour of Robin Milner, G. Plotkin, C. Stirling and M. Tofte, editors, MIT Press, 1998.

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

[7] 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.

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

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

[10] P. Caspi and A. Girault and D. Pilaud, *Automatic Distribution of Reactive Systems for Asynchronous Networks of Processors*, IEEE Trans. on Software Engineering, Vol 25-3, page 416-427, 1999

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

[12] A. Benvenieste, B. Caillaud, P. Le Guernic, *Compositionality in dataflow synchronous languages: specification and distributed code generation*, Information and Computation, vol.163, no.1, 25 Nov. 2000, pp.125-71. Publisher: Academic Press, USA.

[13] M. Zennaro, R. Sengupta *Distributing Synchronous Systems with Modular Structure*, IEEE 2004 44th Conference on Decision and Control, December 2004

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

[15] Jayadev Misra, *Distributed discrete-event simulation*, ACM Computing Surveys (CSUR), Volume 18 Issue 1 March 1986

[16] L. P. Carloni, K. L. McMillan, A. L. Sangiovanni-Vincentelli, *Theory of Latency-Insensitive Design*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems., 20(9):18, September 2001.

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

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

[19] G. Kahn, *The Semantics of a Simple Language for Parallel Programming*, Proceedings of the IFIP Congress74. North Holland Publishing Company.

[20] G. Khan and D.B.MacQueen, *Coroutines and networks of parallel processes*, Information Processing, North-Holand Publishing Co. 1977

[21] E. A. Lee, T. M. Parks, *Dataflow process networks*, Proceedings of the IEEE, 1987

[22] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, *from Simulink to Scade/Lustre to TTA: a layered approach for distributed embedded applications*, ACM LCTES 2003

[23] J. Romberg, A. Bauer, *Loose Synchronization of Event-Triggered Networks for Distribution of Synchronous Programs*, ACM EMSOFT 2004

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

[25] Zennaro, Sengupta, *Distributing Synchronous Programs Using Bounded Queues, a coordinated traffic signal application*, University of California at Berkeley, Intelligent Transportation Studies, UCB-ITS-RR-2005-4, May 2005

[26] H.B. Enderton, *A Mathematical Introduction to Logic*, Academic Press; 2 edition (December, 2000)

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

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

[29] http://www.scoot-utc.com

[30] http://www.its.dot.gov/initiatives/initiative9.htm

[31] J. H. Kell, *Coordination of fixed-time traffic signal*, J. H. K. and Associates Internal report, 1973

[32] M. Boydstun, *Coordinated Traffic Signal Systems*, National Institute for Advanced Transportation Technology, Traffic Signal Summer Workshop, 2004

[33] D. Gitelson, *Traffic Signal Computers*, California Division of Highways Internal report, 1972

[34] M. Zennaro, J. Misener *A State Map Architecture for Safe Intelligent Intersections*, ITS America 2003 13th annual meeting, May 2003

[35] C. F. Daganzo, *Fundamentals of transportation and traffic operation*, Pergamon Edition, 1997

[36] *Simulink Help Manual: Writing S-functions*, MathWorks edition, 2002

[37] G. C. Sih, E. A. Lee, *A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures*,IEEE Transactions on Parallel and Distributed Systems, vol.4, no.2, Feb. 1993, pp.175-87. USA.