Title
Modular Composition of Synchronous Programs: Applications to Traffic Signal Control

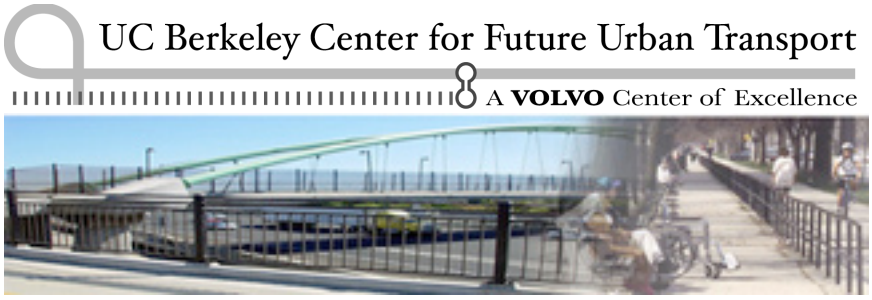
Authors
Zennaro, Marco
Sengupta, Raja

Publication Date
2006-06-01

Modular Composition of Synchronous Programs: Applications to Traffic Signal Control

Marco Zennaro and Raja Sengupta

WORKING PAPER
UCB-ITS-VWP-2006-1



June 2006

Contents

1 Introduction to Embedded Programming for Transportation Systems

This paper describes a modular compilation scheme for distributed synchronous programming. The approach is first described mathematically and then implemented as a library to distribute Simulink (59). Application of the scheme is illustrated by developing a control system to coordinate traffic signals.

The purpose of the research is to advance programming tools for control of large networked systems. For example, USDOT's Vehicle Infrastructure Integration Initiative (54) is rolling out an ad-hoc wireless infrastructure to create the networked roadway. This will enable distributed operation of the traffic control infrastructure. Hence our study of the signal controls application. Typically, these are large-scale systems with a centralized computing architecture controlling thousands of devices connected to a traffic management center over leased telephone lines. For example, the LADOT ATCS system integrates 1300 signals and multiple changeable message signs. The Caltrans I-5/I-405 freeway management system integrates a vast sensor system containing thousands of inductive loops and hundreds of cameras (49). Other areas familiar to the authors that motivate this paper are control for collaborating unmanned air vehicle systems (43) and roadside vehicle systems for crash avoidance (50). The control engineers working on these systems are almost always familiar with Simulink and typically write the first compilable specification of control in the language. We try to extend Simulink through the compilation scheme in this paper to program networked control systems because it is well established as a high-level specification language in the control community.

The compilation scheme relies on *modularity* and *separate code compilation* to help with the size challenge and on *distributed synchronous programming* to handle concurrency and synchronization. Modularity was one of the first programming features introduced by computer scientists to deal with large systems. The idea behind modularity is to extend the *divide et impera* strategy to code generation: the complex code is structured and split into smaller and easier to handle modules. Each module encapsulates a part of the code; it offers some abstract high level services to the rest of the system while hiding unnecessary details. This is the fundamental idea behind the introduction of procedure and objects in modern programming languages.

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 (32),(31),(4),(2). The compiler takes care of translating the synchronous system into sequential code while preserving its semantic (2). Synchronous programming languages like ESTEREL (5)-(6), LUSTRE (21), SIGNAL (25), or Simulink (59) are modular and compositional. When controllers coordinate over networks, both concurrency and nondeterminism are enhanced, due to the asynchronous nature of the communication medium. In the synchronous philosophy, the increased complexity should be hidden from the user by handling it automatically in compilation. This is now an active field of research and it is targeted by this paper.

(11)-(19) 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 (3). This approach preserves the synchronous semantics but does not maintain or exploit the modular structure in the original synchronous program. For example the module structure in the original Esterelle program is lost in the compiled code produced in the approach presented in (11). Consequently, modification to one module of the synchronous program may require re-compilation and re-distribution of the entire system. As the system grows in size, component updates are more frequent. As a result this approach is no longer practicable.

The research presented in this paper tries to alleviate the global compilation problem by investigating a 2 or multi-step compilation approach in which the first step deals with the data dependencies between modules and the second step with timing as done in (23). This paper shows the first step can preserve the modular structure of the synchronous program in its compiled sequential, asynchronous, semantic preserving, equivalent. Any modification to a module of the synchronous program will only require recompilation of the altered module. In this phase the code is annotated with node running time information and with the causality dependencies between its inputs and outputs. For example the module in figure 1 has output 1 depending on inputs 1 and 2, and it executes in 10 ms.

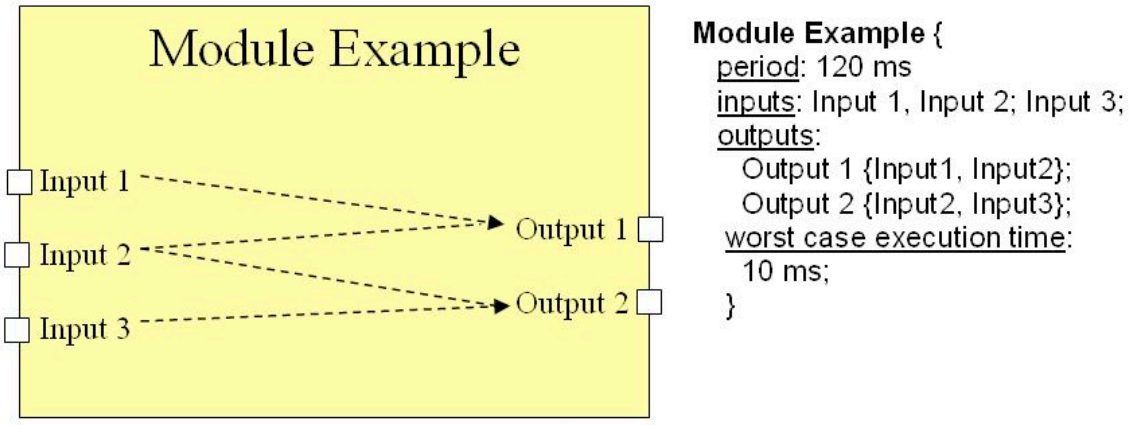


Figure 1: A graphical representation of a Simulink module and the annotation of its compiled equivalent

(3) 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 (51) we proposed such an algorithm based on CSP style rendezvous (24). In this paper we present a bounded queue composition algorithm, its correctness proof, its application to distribute Simulink and its use to develop a control system for coordinated traffic signals. The mathematical results without proof first appeared in (52).

The approach presented in this paper is most similar to (39) and (9). In (39) 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 (9) microcircuit components are composed together under the assumption that they are "*stallable*", and the communication between components is modeled using fix sized FIFO queues.

The second compilation step fulfilling timing specifications on the target hardware, may be different for different hardware architectures. In (23) a hard real-time program can be executed on a platform only if there is a feasible schedule for it. The compiler proves that the program can be executed on the target architecture generating a schedule. The compiler starts with Giotto code annotated similarly to what is proposed here. Using the same set of annotations the algorithm proposed in (33) can be used to check if the input/output

dependencies of the blocks are consistent or if they lead to a deadlock.

Modularity at the second step of compilation may be achievable for certain hardware architectures and not for others. Here we use a second step compilation adequate only for the signal control problem. We compile to execute over Pentium machines connected using TCP/IP sockets. The computation cycles in signal control are in the ten's of seconds. Computation and communication occur so fast that we simply compile to constraint concurrency across modules minimally, i.e., only enough to respect their input-output data dependencies, and then have each module compute and communicate as fast as possible. There is no scheduler. Thus the second step remains modular. The synchronous program compilation scheme in (12) schedules computation and communication to guarantee timing properties for the Time Triggered Architecture (TTA) (45). However, it is not modular. Thus our modular compilation results pertain mainly to the first step of compilation.

The distribution method and its properties are first described mathematically, and then turned into programming libraries for Simulink. We only compile programs in which there are no causal loop.

Synchronous programs are modeled using a finitary version of the Synchronous Transition System introduced in (38), modified to resemble Simulink. The formalism used for the compiled sequential asynchronous code is similar to the I/O automata of (36). 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 (26), (27), (29), but we assume that communication queues have bounded size so they can be realized by 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 (3). 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 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 (12) 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 (3).

The first compilation step 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 (42)). The task is carried out in the second compilation step. This step in the implementation presented in this paper merely enables them to compute and communicate as fast as possible over TCP/IP channels.

The paper is organized as follows. Section 2 introduces a formalism for Simulink-like synchronous systems, and one for sequential asynchronous compiled code. The problem is there formulated mathematically. Section 3 presents the compilation scheme, compares it with one used by Simulink, and proves that the map preserves the synchronous semantic. The main theorem supporting the distribution of Simulink programs is then presented. Section 4 introduces the BDSP library and its performances. An application to traffic control is then described in section 5. Section 6 summarizes the results in the paper and describes future work.

2 Basic models

2.1 Synchronous Systems

Several synchronous system formalisms exist in the literature. The basic idea behind all of them is 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.

2.1.1 STS and FSTS

The Synchronous Transition System formalism, was introduced by Manna and Pnueli in (38). 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:

1.a S is the finite set of state variables of the system.
1.b I is the finite set of input ports of the system. I and S are required to be disjoint.
1.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 2.1.2).
1.d $\sigma_0(S)$ is the initial valuation of the state variables. $\sigma_0(s)$ denotes the initial value of the variable $s \in S$.
1.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.
1.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. 1.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:

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

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 (2.1.2).

2.1.2 FSTS examples

Consider the simple Simulink system in figure (2.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.

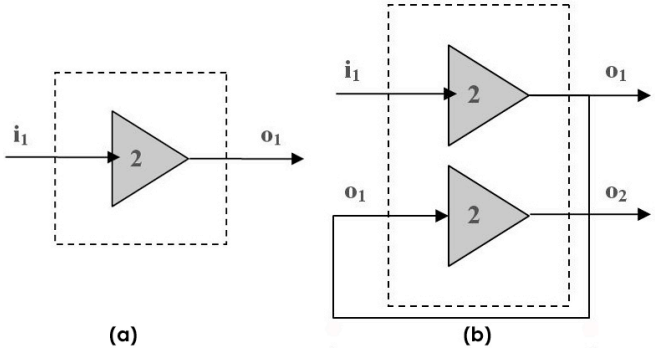


Figure 2: Two examples of Simulink systems

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 (2.a) $I \cap O = \emptyset$. In the example in figure (2.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 $S = \emptyset$, $I = \{i_1, o_1\}$, $O = \{o_1, o_2\}$, $\sigma_0(S) = \emptyset$, $\Psi_S = \emptyset$, $\Psi_O = \{\psi_{o_1} \stackrel{def}{=} 2 * i_1, \psi_{o_2} \stackrel{def}{=} 2 * o_1\}$, $\prec = \{(i_1, o_1), (o_1, o_2)\}$.

2.1.3 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 (see for example (19)), while others have looked for a fixed-point solution (as done in (15)). 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.

2.1.4 Compatible FSTS composition

In this section a composition operator for FSTS is defined. Once again, this is chosen 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:

5.a $O^{s_1} \cap O^{s_2} = \emptyset$,
5.b $S^{s_1} \cap S^{s_2} = \emptyset$,
5.c $S^{s_1} \cap (O^{s_2} \cup I^{s_2}) = \emptyset$,
5.d $S^{s_2} \cap (O^{s_1} \cup I^{s_1}) = \emptyset$,
5.e $I^{s_1} \cap I^{s_2} = \emptyset$,
5.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 non-determinism). 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 2.1.2). 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:

6.a $I = (I^{s_1} \cup I^{s_2})$,
6.b $P_O = (O^{s_1} \cup O^{s_2})$,
6.c $P_S = (S^{s_1} \cup S^{s_2})$,
6.d $\sigma_0(S) = (\sigma_O^{s_1}(S^{s_1}) \cup \sigma_O^{s_2}(S^{s_2}))$,
6.e $\Psi_O = \Psi_O^{s_1} \cup \Psi_O^{s_2}$,
6.f $\Psi_S = \Psi_S^{s_1} \cup \Psi_S^{s_2}$,
6.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 section (2.1.6).

2.1.5 Properties of FSTS composition

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. The result first appeared with no

proof in (52).

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

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. ■

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

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

Proof:

We first prove $\Rightarrow$ by contradiction. Assume that:

$$\begin{array}{c} s_1 \times_{FSTS} s_2 \models t \ \wedge \\ (s_1 \not\models t|P^{s_1} \vee s_2 \not\models t|P^{s_2}) \end{array}$$

It follows by the definition (4) of trace satisfaction, that:

$$\forall j \in \mathbb{N}_t \quad s_1 \times s_2 \models t_j \ \wedge \tag{2}$$

$$\exists i \in \mathbb{N}_t \quad s_1 \not\models t_i|p^{s_1} \vee s_2 \not\models t_i|p^{s_2} \tag{3}$$

Now pick the smalles i for which (3) holds. There are two possible cases. Either $i = 0$ or $i > 0$.

Case $i > 0$: By definition (3) of tuple satisfiability and by (2) it follows that $\exists p \in (O^{s_1} \cup O^{s_2} \cup S^{s_1} \cup S^{s_2})$ such that:

$$\begin{array}{rl} t_i|p = \psi_p^{s_1 \times s_2}(t_i|P_p^{s_1 \times s_2}) & \text{if } p \in O^{s_1 \times s_2} \\ t_i|p = \psi_p^{s_1 \times s_2}(t_{i-1}|P_p^{s_1 \times s_2}) & \text{if } p \in S^{s_1 \times s_2} \end{array} \tag{4}$$

By definition (3) of tuple satisfiability and by (3) it follows that $\exists p \in (O^{s_1} \cup O^{s_2} \cup S^{s_1} \cup S^{s_2})$ such that:

$$\begin{array}{rl} t_i|p \neq \psi_p^{s_1}(t_i|P_p^{s_1}) & \text{if } p \in O^{s_1} \\ t_i|p \neq \psi_p^{s_1}(t_{i-1}|P_p^{s_1}) & \text{if } p \in S^{s_1} \\ t_i|p \neq \psi_p^{s_2}(t_i|P_p^{s_2}) & \text{if } p \in O^{s_2} \\ t_i|p \neq \psi_p^{s_2}(t_{i-1}|P_p^{s_2}) & \text{if } p \in S^{s_2} \end{array} \tag{5}$$

For the minimal i pick a minimal port for which conditions (2-3) hold with respect to $\prec_{s_1 \times s_2}$. Denote this minimal port by p. We assume that $p \in P_O^{s_1 \times s_2}$, the case $p \in P_S^{s_1 \times s_2}$ has a similar proof.

By definition of FSTS composition it follows that either $p \in O^{s_1}$ or $p \in O^{s_2}$. Assume that $p \in O^{s_1}$. The proof for $p \in O^{s_2}$ is the same up to a change of superscipt. Now:

$$\begin{array}{rll} t_i|p = & \psi_p^{s_1 \times s_2}(t_i|P_p^{s_1 \times s_2}) & \text{from (4)} \\ = & \psi_p^{s_1}(t_i|P_p^{s_1}) & \text{by def. of FSTS comp.} \end{array} \tag{6}$$

But this contradict (5).

Case $i = 0$: By definition (3) of tuple satisfiability and by (2) the following must hold: $\exists p \in (O^{s_1} \cup O^{s_2} \cup S^{s_1} \cup S^{s_2})$.

$$\begin{aligned} t_i|p &= \sigma_0^{s_1 \times s_2} && \text{if } p \in S^{s_1 \times s_2} \\ t_i|p &= \psi_p^{s_1 \times s_2}(t_i|P_p^{s_1 \times s_2}) && \text{if } p \in O^{s_1 \times s_2} \end{aligned} \tag{7}$$

By definition (3) of tuple satisfiability and by (3) the following must hold: $\exists p \in (O^{s_1} \cup O^{s_2} \cup S^{s_1} \cup S^{s_2})$.

$$\begin{aligned} t_i|p &\neq \sigma_0^{s_1}(p) && \text{if } p \in S^{s_1} \\ t_i|p &\neq \psi_p^{s_1}(t_i|P_p^{s_1}) && \text{if } p \in O^{s_1} \\ t_i|p &\neq \sigma_0^{s_2}(p) && \text{if } p \in S^{s_2} \\ t_i|p &\neq \psi_p^{s_2}(t_i|P_p^{s_2}) && \text{if } p \in O^{s_2} \end{aligned} \tag{8}$$

For $i = 0$, pick a minimal port for which the above conditions hold with respect to $\prec_{s_1 \times s_2}$ and denote it p. If $p \in O^{s_1 \times s_2}$, we can follow the same proof as the previous case. Therefore let $p \in S^{s_1 \times s_2}$. Assume that $p \in P_S^{s_1}$. The proof for the case $p \in S^{s_2}$ is the same up to a change of superscript.

By definition of FSTS composition, given the assumption $p \in S^{s_1}$, from (7) follows that:

$$t_i|p = \sigma_0^{s_1}(p).$$

But this contradicts (8).

We now prove the second implication $\Leftarrow$ by contradiction. Assume that:

$$\begin{gathered} s_1 \times_{FSTS} s_2 \not\models t \ \wedge \\ (s_1 \models t|P^{s_1} \wedge s_2 \models t|P^{s_2}) \end{gathered}$$

It follows by the definition (4) of trace satisfaction that:

$$\forall j \in \mathbb{N} \quad s_1 \models t_j|p^{s_1} \wedge s_2 \models t_j|p^{s_2} \ \wedge \tag{9}$$

$$\exists i \in \mathbb{N} \quad s_1 \times s_2 \not\models t_i \tag{10}$$

Now pick i to be the smallest number for which (10) holds. There are two possible cases. Either $i = 0$ or $i > 0$.

Case $i > 0$: By definition (3) of trace satisfaction and by (9) it follows that: $\exists p \in (O^{s_1} \cup O^{s_2} \cup S^{s_1} \cup S^{s_2})$.

$$\begin{aligned} t_i|p &= \psi_p^{s_1}(t_i|P_p^{s_1}) && \text{if } p \in O^{s_1} \\ t_i|p &= \psi_p^{s_1}(t_{i-1}|P_p^{s_1}) && \text{if } p \in S^{s_1} \\ t_i|p &= \psi_p^{s_2}(t_i|P_p^{s_2}) && \text{if } p \in O^{s_2} \\ t_i|p &= \psi_p^{s_2}(t_{i-1}|P_p^{s_2}) && \text{if } p \in S^{s_2} \end{aligned} \tag{11}$$

$$\tag{12}$$

By definition (4) of trace satisfaction and by (10) it follows that: $\exists p \in (O^{s_1} \cup O^{s_2} \cup S^{s_1} \cup S^{s_2})$.

$$\begin{aligned} t_i|p &\neq \psi_p^{s_1 \times s_2}(t_i|P_p^{s_1 \times s_2}) && \text{if } p \in O^{s_1 \times s_2} \\ t_i|p &\neq \psi_p^{s_1 \times s_2}(t_{i-1}|P_p^{s_1 \times s_2}) && \text{if } p \in S^{s_1 \times s_2} \end{aligned} \tag{13}$$

For the minimal i for which conditions (10) holds, pick a minimal port for which conditions (9-10) hold with respect to $\prec_{s_1 \times s_2}$. Denote this minimal port as p. We assume that $p \in O^{s_1 \times s_2}$. The case $p \in S^{s_1 \times s_2}$ has a similar proof.

By definition of FSTS composition it follows that either $p \in O^{s_1}$ or $p \in O^{s_2}$. Assume that $p \in O^{s_1}$ (the proof for $p \in O^{s_2}$ is the same up to a change of superscipt). Now:

$$\begin{aligned} t_i|p \neq & \quad \psi_p^{s_1 \times s_2}(t_i|P_p^{s_1 \times s_2}) && \text{from (13)} \\ = & \quad \psi_p^{s_1}(t_i|P_p^{s_1}) && \text{by def. of FSTS comp.} \end{aligned} \tag{14}$$

But this contradicts (11).

Case $i = 0$: By definition (4) of tuple satisfiability and by (10) follows that: $\exists p \in (O^{s_1} \cup O^{s_2} \cup S^{s_1} \cup S^{s_2})$.

$$\begin{aligned} t_i|p = \sigma_0^{s_1}(p) & \quad \text{if } p \in S^{s_1} \\ t_i|p = \psi_p^{s_1}(t_i|P_p^{s_1}) & \quad \text{if } p \in O^{s_1} \\ t_i|p = \sigma_0^{s_2}(p) & \quad \text{if } p \in S^{s_2} \\ t_i|p = \psi_p^{s_2}(t_i|P_p^{s_2}) & \quad \text{if } p \in O^{s_2} \end{aligned} \tag{15}$$

$\exists p \in (O^{s_1} \cup O^{s_2} \cup S^{s_1} \cup S^{s_2})$.

$$\begin{aligned} t_i|p \neq \sigma_0^{s_1 \times s_2}(p) & \quad \text{if } p \in S^{s_1 \times s_2} \\ t_i|p \neq \psi_p^{s_1 \times s_2}(t_i|P_p^{s_1 \times s_2}) & \quad \text{if } p \in O^{s_1 \times s_2} \end{aligned} \tag{16}$$

For $i = 0$ pick a minimal port for which the above conditions hold with respect to $\prec_{s_1 \times s_2}$. Denote this port with p. If $p \in O^{s_1 \times s_2}$, we can follow the same proof as the previous case. Therefore $p \in S^{s_1 \times s_2}$. Assume that $p \in S^{s_1}$. The proof for the case $p \in S^{s_2}$ is the same up to a change of superscript. By definition of FSTS composition, given the assumption $p \in S^{s_1}$, from (16) follows that:

$$t_i|p \neq \sigma_0^{s_1}(p)$$

but this contradict (15).

This conclude the proof.

■

2.1.6 FSTS composition examples

Consider the Simulink system in figure (3.a). The system is composed of two blocks similar to the one described in section 2.1.2. 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

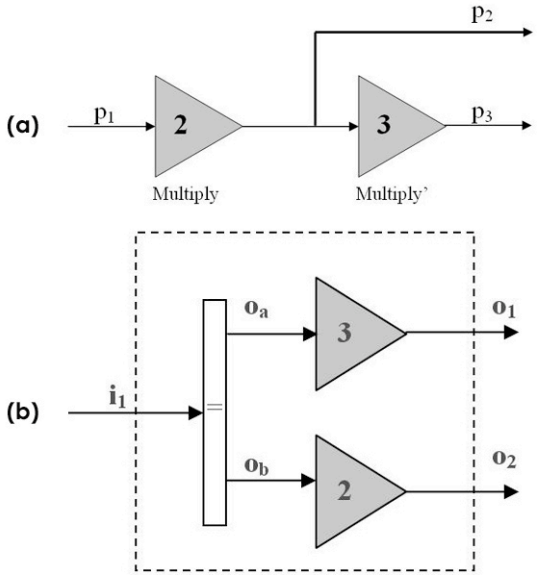


Figure 3: Simulink systems composed of multiple blocks

in figure (3.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: $I = \{i_1, o_a, o_b\}$, $O = \{o_a, o_b, o_1, o_2\}$, $S = \emptyset$, $\sigma_0(S) = \emptyset$, $\Psi_S = \emptyset$, $\Psi_O = \{\psi_{o_a} \stackrel{def}{=} i_1, \psi_{o_b} \stackrel{def}{=} i_1, \psi_{o_1} \stackrel{def}{=} 3 * o_a, \psi_{o_2} \stackrel{def}{=} 3 * o_b\}$, $\prec = \{(i_1, o_a), (i_1, o_b), (o_a, o_1), (o_b, o_2)\}$.

2.2 Asynchronous Systems

There are many asynchronous system formalisms in the literature. One of them is the asynchronous version of STS, called the Asynchronous Transition System (ATS) model, introduced by Benvenieste in (4). 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 the 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 (36) 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 (16)), 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 (4) and is fomalized as the following RA:

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

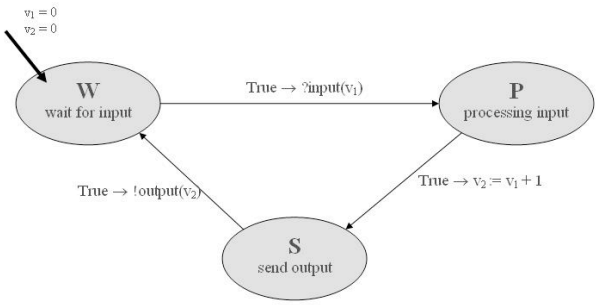


Figure 4: A simple reactive automaton

2.2.1 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 (4) for the input port valuation $\{1\}$ and the output port valuation $\emptyset$:

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

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:

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

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:
$act(a, \sigma(p_1), .., \sigma(p_m), \sigma(v_1), .., \sigma(v_n)) =$

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

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

Proof: Follows from the associativity and commutativity of the union operator, and the fact that the identity element of the union operator is the empty set. Please not that the empty RA is the identity element in the sense that, if composed with an automaton w, the reulting automaton is bisimilar to w. ■

$\prod_{w \in W} w$ denotes an n-ary composition of RA's. Lemma (2.3) 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:

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

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

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

Proof: Follows from the observation that every r_i in r belongs to $r|w_1$ or to $r|w_2$. This is so because the transition in each tuple belongs to one of the two automata or it is an ϵ action. If the action belongs to $r|w_1$, by definition of RA composition, it does not modify the location, variables or output ports of w_2 and viceversa. ■

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 2.5. *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 (22)), as well as with 3-way handshakes protocols over a network (see (47)). This means we can compose RAs located at different sites across networks. In section 4 we will explore an approach that takes full advantage of the distribution of the code (maximising pipeline gain).

2.3 Problem formulation

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 χ :$\mathbb{T}_{RA} \rightarrow \mathbb{T}_{FSTS}$ where $\mathbb{T}_{RA}$ and $\mathbb{T}_{STS}$ are the set of traces of STS and RA respectively. In (3) 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 ϕ :$\mathbb{F}$STS$\rightarrow\mathbb{R}$A such that the following holds:

$$\forall w \in R_A\, \forall s \in F_{STS}\, .\, w = \phi(s) \Rightarrow (r \models t \Leftrightarrow s \models \chi(t)) \tag{17}$$

If this holds then ϕ maps a synchronous system into an asynchronous system while preserving the synchronous semantic. It has been proved in (3) that for the set of *endochronous* programs such a ϕ exists. In section 3 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 (2). Given such maps we can now formulate our problem (similarly to what is done in (3)) 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:

$$\begin{array}{c} w_1 = \phi(s_1)\ \wedge w_2 = \phi(s_2)\ \Rightarrow \\ (w_1 \times_{RA} w_2 \models t \Leftrightarrow s_1 \times_{STS} s_2 \models \chi(t)) \end{array} \tag{18}$$

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 (3) that when the pair (s_1, s_2) is *isochronous* than such an operator exists. In section 3 we prove that property (18) holds if the two synchronous system are *compatible* (as defined in section 2.1). Thus we claim ϕ is a monomorphism between ($\mathbb{F}$STS, $\times_{FSTS}$) and ($\mathbb{R}$A, $\times_{RA}$).

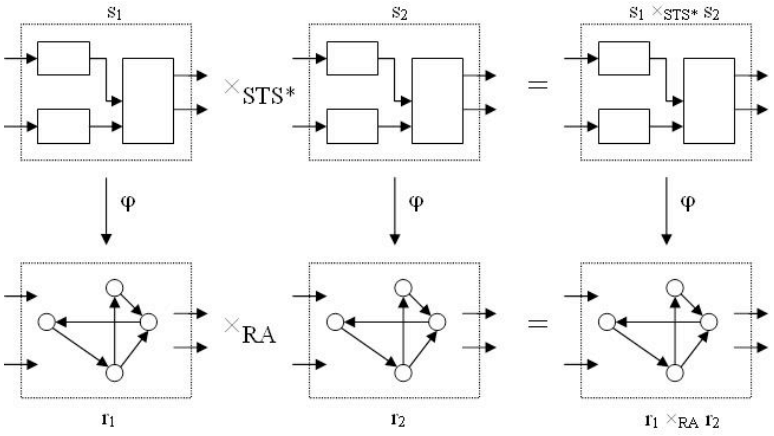


Figure 5: A graphical representation of property (18)

3 Theoretical results

3.1 Implementation of FSTS systems

In this section ϕ, a mapping of FSTSs into RAs is given. It is then proven that the ϕ satisfies (17).

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 $(Nodes, Edges)$

```
1  Nodes := {root, leaf}
2  Edges := ∅
3  % max-int is a global variable that holds the highest
   % integer used to label a node
   counter := max − int + 1
4  ∀ linearization w = (w_1, w_2, ..., w_m) of ≺ in P do
5    pointer = root
6    For all i ∈ [1, m] do
7      if (pointer, n, w_i) ∈ Edges do pointer = n
8        else do
9            add n_counter to Nodes
10           add (pointer, n_counter, w_i) to Edges
11           pointer := n_counter
12           counter + +
13         od
14   od
15 od
16 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.

Some lemmas are now proved.

Lemma 3.1. $ComputeGraph(\prec, P, root, leaf)$ *produces an acyclic graph with source, named* $root$*, and sink, named* $leaf$*. Every path in the graph from source to sink has one and only one edge labelled with an element of* P*. Moreover if* $p' \prec p$ *and* $\{p, p'\} \subseteq P$ *then the edge labelled* p' *appears before the one labelled* p *in every path from* $root$ *to* $leaf$*.*

Proof: Every time an edge is added (on line 11), it does not create a loop because it connects an existing node to a new one. Line 17 does not create any loop since it flattens all the sinks into a single sink. Therefore the graph is acyclic, it has a source $root$ and a single sink $leaf$. By construction every path corresponds to a linearization of $\prec$ in V Therefore an element p' of P appears as a label only once in a path and it appears before all the p for which $(p', p) \in \prec$. ■

Lemma 3.2. *For all* w *in* $\phi[FSTS]$ *and every infinite run* r *of* w*,* r *visits the location* l_{leaf} *and* l_{root} *infinitely often.*

Proof: Proof: The automaton generated by algorithm ϕ are obtained linking two graphs generated by $ComputeGraph$, so that the source of one is the sink of the other. The only nodes shared by the two graphs are $root$ and $leaf$. Each graph is acyclic, has finitely many states, one source and one sink by lemma (3.1). Since every run correspond to an infinite length path in the combined graph and the two graphs are acyclic l_{leaf} and l_{root} are visited an infinite amount of times. ■

From lemma (3.2), we see that any run r = $< r_0, r_1, r_2, ... >$ of an RA in $\phi[FSTS]$ has an infinite subsequence $< r_{i_0}, r_{i_1}, r_{i_2}, ... >$ such that $\forall k \in \mathbb{N}\ r_{i_k} | location = l_{leaf}$ and $\forall k \in \mathbb{N}\ r_i \neq r_{i_k} \Rightarrow r_i | location \neq l_{leaf}$.

Thus we can write r equivalently as r = $< u_0, u_1, u_2, ... >$ where $u_0 =< r_0, ..., r_{i_0} >, u_1 =< r_{i_0+1}, ..., r_{i_1} >$, $u_2 =< r_{i_1+1}, ..., r_{i_2} >$ and so on. We call these u_i's *cycles*. We can also define the function $cycle(r, n)$, for a run r and $n \in \mathbb{N}$ as $cycle(r, n) = r_{i_n}|V$, i.e. as the valuation of V at the n^{th} visit to l_{leaf}, where V is the set of variables of the RA.

In the following, the initialization of the state variables is considered the 0^{th} write of the variables.

Lemma 3.3. *Let $w = \phi(s)$. In every cycle all the input ports of w are read once and only once. Similarly all the output ports and all the variables of w are written once and only once every cycle. If $v' \prec v$ in s then v' is written before v in every cycle of w.*

Proof: Proof: Follows from lemma (3.1) and the definition of algorithm ϕ. ■

Lemma 3.4. *For a given run r of an RA $\phi(s) \in \phi(FSTS)$, let t be the associated trace. Then the following holds $\forall i \in \mathbb{N}\ \forall v \in V\ .\ (t|_v)_i = cycle(r, i)|v$. Moreover:*

$$\begin{array}{rl} (\forall i \in \mathbb{N} \ast & \forall v \in O\ .\ (t|_v)_i = cycle(r,i)|v = \psi_v(\chi(t)_i|I_v \cup S_v) \wedge \\ & \forall v \in S\ .\ (t|_v)_{i+1} = cycle(r,i)|v = \psi_v(\chi(t)_i|I_v \cup S_v)) \wedge \\ \forall v \in S\ . & (t|_v)_0 = cycle(r,0)|v = \sigma_0(v) \end{array}$$

Proof: By lemma (3.3) in every cycle a variable is written once and only once before hitting l_{leaf}. When a run hits the location l_{leaf} for the i^{th} time, all the variables have been written exactly i times. Write actions are introduced by ϕ in lines 15, 19 and 23. Every write to v is ψ_v applied to $t|V$. By lemma (3.3) we then get $(t|_v)_i = \psi_v((t|_{V_v})_i)$. The result then follows by definition (15) of χ. ■

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 3.5. *Algorithm ϕ satisfies property (17), i.e.*

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

Proof: First the left to right ($\Rightarrow$) implication is proved by contradiction. Assume that the implication does not hold. Then the following must hold:

$$\exists s \in FSTS,\ \exists t \in \Gamma,\ w = \phi(s) \in RA.\ w \models t \wedge s \not\models \chi(t)$$

where Γ is the set of traces of w. Let s = $(S, I, O\sigma_0(S), \Psi_O, \Psi_S, \prec)$.

Since $s \not\models \chi(t)$, by definition of FSTS satisfiability, the following must hold:

$$\exists i \in \mathbb{N}\ .s \not\models \chi(t)_i$$

By definition 4 and 5 of FSTS satisfaction it follows that: $\exists i \in \mathbb{N}\ .\exists v \in (O \cup S)$.

$$v \in O\ \Rightarrow\ \chi(t)_i|v \neq \psi_v(\chi(t)_i|I_v \cup S_v) \wedge \tag{19}$$

$$v \in S\ \wedge\ i > 0\ \Rightarrow\ \chi(t)_i|v \neq \psi_v(\chi(t_{i-1})|I_v \cup S_v)$$

$$v \in S\ \wedge\ i = 0\ \Rightarrow\ \chi(t)_i|v \neq \sigma_0(v) \tag{20}$$

Assume $v \in O^s$ (the proof for the case $v \in S^s$ is similar). Since by hyphotesis $w \models t$, by lemma (3.4) the following holds:

$$\forall i \in \mathbb{N} \,.\, \forall v \in O \,.\, t_i|v = \psi_v(\chi(t)_i|_{I_v \cup S_v}) \tag{21}$$

Pick i to be the minimal for which (19) holds. Pick then v to be one of the minimal (with respect to $\prec^s$) variables for which (19) holds. It then follows from (19) and (21) that:

$$\chi(t)_i|v \neq t_i|v$$

But this contradict definition (15) of χ. Hence the first implication of the theorem holds.

The proof for the right to left ($\Leftarrow$) is now given. It is shown that for any trace t' of an FSTS s there is a run r of $\phi(s)$ with associated trace t such that $t' = \chi(t)$.

Consider the run r constructed cycle by cycle as follows. Fix any linearization of $\prec |_{O \cup I}$. This linearization correspond to an unique path from l_{root} to l_{leaf}, where each edge label updates a variable in the order given by the linearization. Extend the linearization with the total order defined on line (22) of ϕ so that the state variables in S follow all the others in the order. This order fixes now a unique cycle from l_{root} back to itself, where by lemma (3.3) each variable is updated once and only once and in the order given by this extended linearization.

Coonsider the run starting from location l_0 with all the variables initialized to $\sigma_0(V)$. At the begining of every cycle, through through ϵ transition, all the output ports are emptied and all the external inputs are supplied. Then the run goes through the cycle identified by the selected linearization. The value written in each ports p_v in the i^{th} cycle is $(t'_i)|_v$. The value written in each variable v in the i^{th} cycle is $(t'_i)|_v$.

This run is associated by construction to a trace t such that $t' = \chi(t)$. We need to show that it satisfies w. By lemma (3.4) the values of the variables and of the ports are the ones satisfying w.

It is left to show that that w would not deadlock at any point of the run. All the reactive automata generated through ϕ have no sink states, and since all their transitions have only $true$ guards, there is always a transition enabled. This means that the execution of w can be blocked only on a read from an empty input queue or a write on a full output queue.

The external input ports are written at the begining of each cycle of the constructed run and, by lemma (3.3), the queue is then read once and only once so there are no blocking reads on an external input queue. At the end of the cycle the queue is empty preventing writes on full queues at the begining of the next cycle.

The external output ports are emptied at the begining of each cycle of the constructed run and, by lemma (3.3), they are written once and only once per cycle. Hence there are no blocking writes on external output queues.

This conclude our proof. ■

3.2 Implementation of Simulink systems

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.

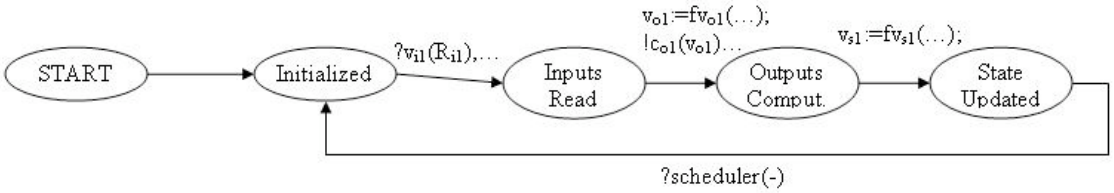


Figure 6: Implementation of a Simulink System

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}.

3.3 Distribution of FSTS systems

We have seen in the previous section that there is a map ϕ between FSTS and RA satisfying property (17). We now prove that the composition operator $\times_{RA}$ as introduced in section 2.2 satisfies property (18).

Since two different RAs may be running on different machines, they 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 been valuated before v'. For the class of reactive automata implementing an FSTS, i.e. $\psi[FSTS]$ this is formalized by the following observation:

Proposition 3.6. *Consider two compatible RA $w_1 = \phi(s_1)$ and $w_2 = \phi(s_2)$ with variables v_1, v_3 of w_1 and w_2 respectively and a port p_2 written by w_1 and read by w_2. If in each cycle of w_1, v_1 is written before p_2 is written and in each cycle of w_2, p_2 is read before v_3 is written by w_2, then v_1 is written for the i^{th} time after v_3 is written for the i^{th} time in $w_1 \times_{RA} w_2$.*

Proof: Since the two RA are compatible only one automaton can write on any port. By hyphothesis only w_1 writes on p_2 and by (2.5) no messages are lost. Hence, since every read operation removes an element from the queue and that the queues are initially empty, for w_2 to be reading from p_2 $(i)^{th}$ times, w_1 must have written p_2 $(i)^{th}$ times. By hyphothesis for w_2 to be writing v_3 for the i^{th} time, it must have read p_2 i^{th} times. Thus, by hyphothesis on w_1 v_1 has been written at least i^{th} times. ■

We have claimed in section 2.2 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:

Theorem 3.7. *The compostion operator $\times_{RA}$ satisfies property (18), i.e. for any two compatible FSTS s = (S^s, I^s, O^s, I^s_O, Ψ^s_O, Ψ^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:*

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

Proof: The theorem is proved proving the two implications separately, starting with the left to right ($\Rightarrow$) implication, now proved by contradiction. Assume that the thesis does not hold. Than the $\exists s, s' \in FSTS, \exists w, w' \in RA, \exists t \in \Gamma^{r \times r'}$.

$$w = \phi(s) \wedge w' = \phi(s') \wedge \tag{22}$$

$$(w \times_{RA} w') \models t \wedge \tag{23}$$

$$(s \times_{STS^*} s') \not\models \chi(t) \tag{24}$$

where $\Gamma^{w \times w'}$ denotes set of traces of $w \times_{RA} w'$. Assume $w = (L^w, l_0^w, V^w, V_0^w, P_I^w, P_O^w, T^w)$ and $w' = (L^{w'}, l_0^{w'}, V^{w'}, V_0^{w'}, P_I^{w'}, P_O^{w'}, T^{w'})$,

From the definition (4) of trace satisfaction (24) is equivalent to:

$$\exists i \in \mathbb{N} \,.\, (s \times_{FSTS} s') \not\models \chi(t)_i$$

Pick the smallest i for which the above condition holds and denote it with i. From definition (3) of tuple satisfaction it then follows that:

$$\exists v \in (S^{s \times s'} \cup O^{s \times s'}) \,.\, \chi(t)_i|_v \neq \psi_v^{s \times s'}(\chi(t)_i|_{(I_v^{s \times s'} \cup S_v^{s \times s'})}) \tag{25}$$

Amongst the variables at i satisfying (25) pick a minimal one w.r.t. $\prec_{s \times s'}$, and denote it v, Assume that $v \in O^{s \times s'}$. The proof for the case $v \in S^{s \times s'}$ is similar. Assume that in particular $v \in O^s$. The proofs for the case $v \in O^{s'}$ is the same up to a superscript.

Now by contradiction hyphothesis (23) and lemma (2.4) the following hold:

$$w \models t$$

Hence, by lemma (3.4) and by the assumption the following hold:

$$\forall k \in \mathbb{N} \, \forall y \in O^s \,.\, (t|_y)_k = \psi_y^s(\chi(t)_k|_{(I_y^s \cup S_y^s)}) \tag{26}$$

In particular this holds for $y = v$ and $k = i$. So that:

$$\begin{aligned}
\chi(t)_i|_v &= (t|_v)_i && \text{by definition } (\chi) \text{ of } \chi \\
&= \psi_v^s(\chi(t)_i|_{(I_v^s \cup S_v^s)}) && \text{from (26) by lemma (3.4)} \\
&= \psi_v^{s \times s'}(\chi(t)_i|_{I_v^{s \times s'} \cup S_v^{s \times s'}} && \text{by def. of FSTS comp.}
\end{aligned}$$

But this contradict (25) hence the first implication is proved.

The proof of the right to left implication ($\Leftarrow$) is now given. It is shown that for any trace t' of an FSTS $s \times s'$ there is a run r of $\phi(s) \times \phi(s')$ with associated trace t such that $t' = \chi(t)$.

Consider the run r constructed cycle by cycle as follows. Let $E = ((O^{s \times s'} \backslash I^{s \times s'}) \cup (I^{s \times s'} \backslash O^{s \times s'}))$ be the set of external input and ouput ports. Fix a linearization of $\prec |_E$. This linearization, projected on the ports of s identifies an unique path from l_{root} to l_{leaf} in $\phi(s)$. Similarly when projected on the ports of s', it identifies an unique path from l_{root} to l_{leaf} in $\phi(s')$. In both cases each label of each edge of the paths updates a variable in the order given by the linearization.

Extend the linearization with the total orders defined on line (22) of phi. Then the state variables in $S^{s\times s'}$ follow all the other variables in the order. The orders fix a unique cycle from l_{root} back to itself in both $\phi(s)$ and $\phi(s')$, where by lemma (3.3) each variable is updated once and only once and in the order given by the selected total order.

The run starts from location $\{(\phi(s), l_0^s), (\phi(s'), l_0^{s'})\}$ with all the variables initialized to $\sigma_0(V)$ and $\sigma_0^{s'}(V)$. At the begining of every cycle, through ϵ transitions, all the external output ports are emptied and all the external inputs are given. Then the run goes through the two automata along the paths identified by the just constructed linearization. The value written in each ports p_v in the i^{th} cycle is $(t'_i)|_v$. The value written in each variable v in the i^{th} cycle is $(t'_i)|_v$.

This run is associated by construction to a trace t' such that $t' = \chi(t)$. We need to show that it satisfies $\phi(s) \times \phi(s')$. By lemmas (2.4-3.4) the values of the variables and of the ports are the ones satisfying $\phi(s) \times \phi(s')$.

It is left to show that that $\phi(s)$ and $\phi(s')$ would not deadlock at any point of the run. All the reactive automata generated through ϕ have no sink states, and since all their transitions have only $true$ guards, there is always a transition enabled. This means that the execution of $\phi(s)$ and $\phi(s')$ can be blocked only on a read from an empty input queue or a write on a full output queue.

The external input ports are written at the begining of each cycle and, by lemma (3.3), the queue is then read once and only once so there are no blocking reads on an external input queue. At the end of the cycle the queue is empty preventing writes on full queues at the begining of the next cycle.

The external output ports are emptied at the begining of each cycle of the constructed and, by lemma (3.3), they are written once and only once per cycle. Hence there are no blocking writes on external output queues.

The only remaining blocking condition possible is on internal input (i.e. ports that belongs to $(I^s \cap O^{s'}) \cup (I^{s'} \cap O^s)$). Since they are not internal inputs these ports are empty at the begining of every cycle. They are written once by one automaton and read once and only once by the other. Hence they are empty at the end of each cycle.

By lemma (3.6) the write action take place before the read action. Thus there is no blocking read or write on internal inputs.

This concludes our proof.

■

3.4 Distribution of Simulink systems

As noted in section 3.1 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 3.7 do not extend in the general case for ϕ_{sim}. It suffices to consider the FSTS in figure (7) (taken from (3)).

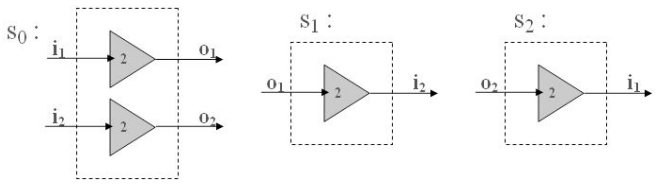


Figure 7: Three FSTS systems

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 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 3.8. *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. ■

4 Tools for the modular distribution of Synchronous Programs

4.1 BDSP architecture

In this section the software architecture for the distribution of Simulink programs (see figure (8)) 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 and was described in (52). A second version, utilising bounded queues as described in this section is currently available as a beta version.

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 (8). 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 3.1). 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).

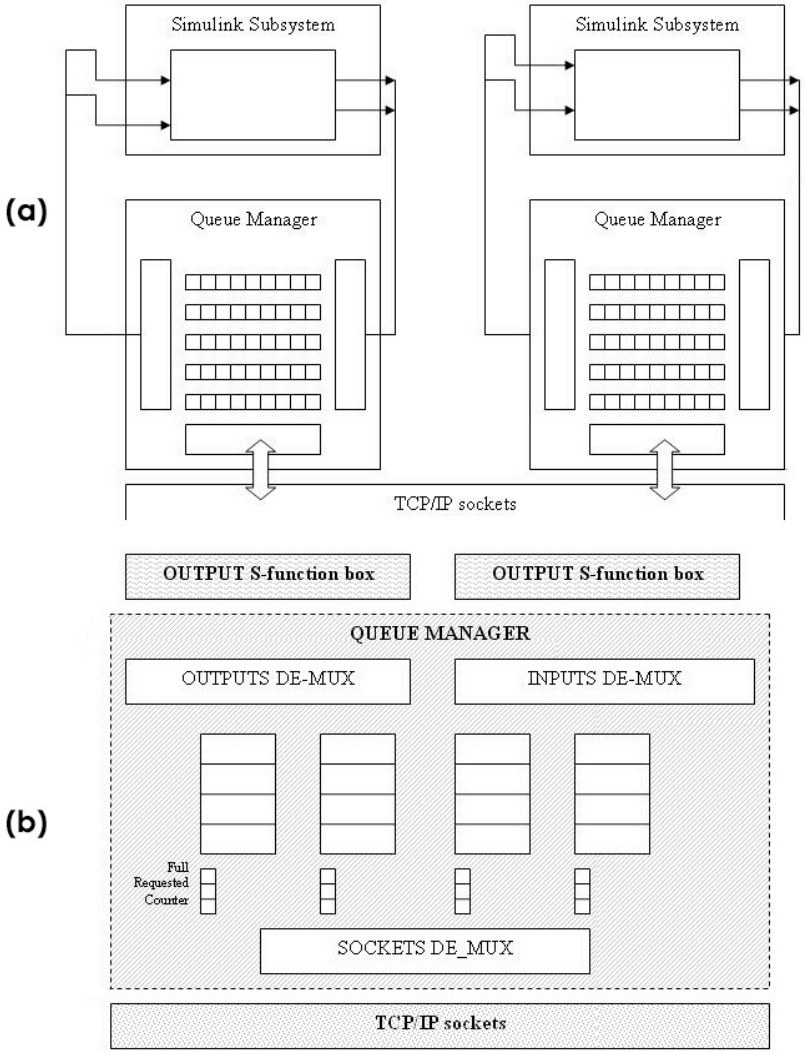


Figure 8: BDSP architecture

4.2 Performance analisys

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 4.1.

We decompose the system in figure (9) into three subsystems running on two separate Pentium 4 850 Mhz,

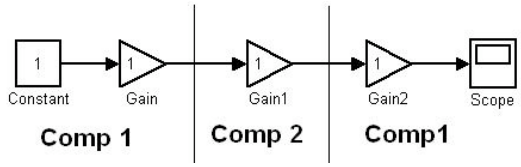


Figure 9: The model used to estimate the overhead

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.

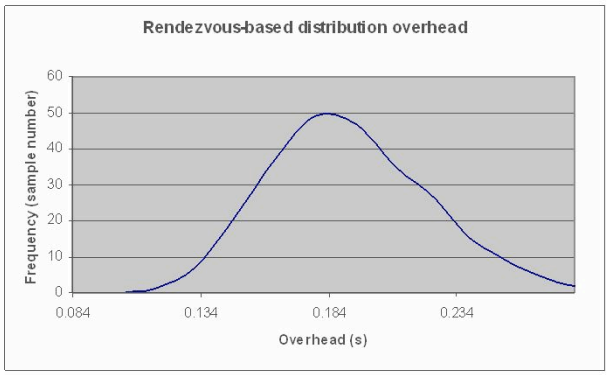


Figure 10: A conservative estimate of the distribution overhead

The results are plotted in figure (10). 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. Even with this overhead the requirements to develop classic traffic control applications are met. In order to use this approach for safety critical applications it is necessary to at least half the overhead. This should be easilly achieved moving from simulation to implementation.

5 Applications to Traffic Signal Control

This section focuses on the problems faced during the development and maintenance of distributed traffic signal control system. Traffic signal control schemes are periodic in nature and are usually expressed using difference equations. Because of it, tools like Simulink seem to be a natural candidate for their development. Modularity preserving approaches are needed because of the frequent system upgrades, traffic network changes and because of the size these systems are reaching. The city of Los Angeles, for example, had more than 4300 traffic controllers operating in 2003. As a case study an off-set control for coordinated traffic lights is developed using Simulink and the BDSP library. The performance of the implemented system is then presented.

5.1 Traffic Signal Control Systems

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

The algorithms used to compute the optimal cycle length and interval splitting can be organized into three categories:

- *pre-timed or fixed time* controllers, based on historical data collected at the intersection;
- *semi-actuated* controllers, that adjusts to side street demands;
- *fully actuated* controllers, that adjusts to both street demands;

The different approaches differ in term of effectiveness and complexity. While the performances of pre-timed systems degrade as the traffic demand deviates from the average one, the actuated approaches compensate for these deviations. At the same time these last approaches are, quoting directly from (7) “extremely difficult to program”.

The complexity of the traffic system increases when multiple traffic signals are coordinated as a signal network. Signal coordination is then used to significantly increase the flow (see (28), (8)). Mainly because of the complexity of the system, coordinated controllers are often pre-timed.

Traffic light operations are traditionally directed by a traffic signal controller, defined in (20) as "a device which controls the flow of traffic at an intersection according to some predetermined rules of operation". 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. It also supports a wide set of sensors and is equipped with multiple communication interfaces (e.g. RS232, Ethernet). It is de-facto a general purpose computer used as a special purpose computer: it supports many predefined rules that can be adjusted on the field or remotely, using the National Transportation Communications for ITS protocol, a.k.a. NTCIP, set of standards as described in (55).

The main problem is that adjustments are possible only to a limited extent. It is not possible to introduce new rules of operation without going back to the manufacturer for a custom design. This increases the cost of the device and its upgrades. Moreover, as pointed out in (20), the custom design product does not usually behave as specified by the traffic engineer.

As pointed out in (20), a possible solution is to replace these devices with general purpose computers, especially in "any intersection that requires concurrent phase-timing or unusual features". This approach is usually the only choice for a researcher seeking to test new rules of operations or new sensor devices (see for example the PATH IDS project, (50), (37)).

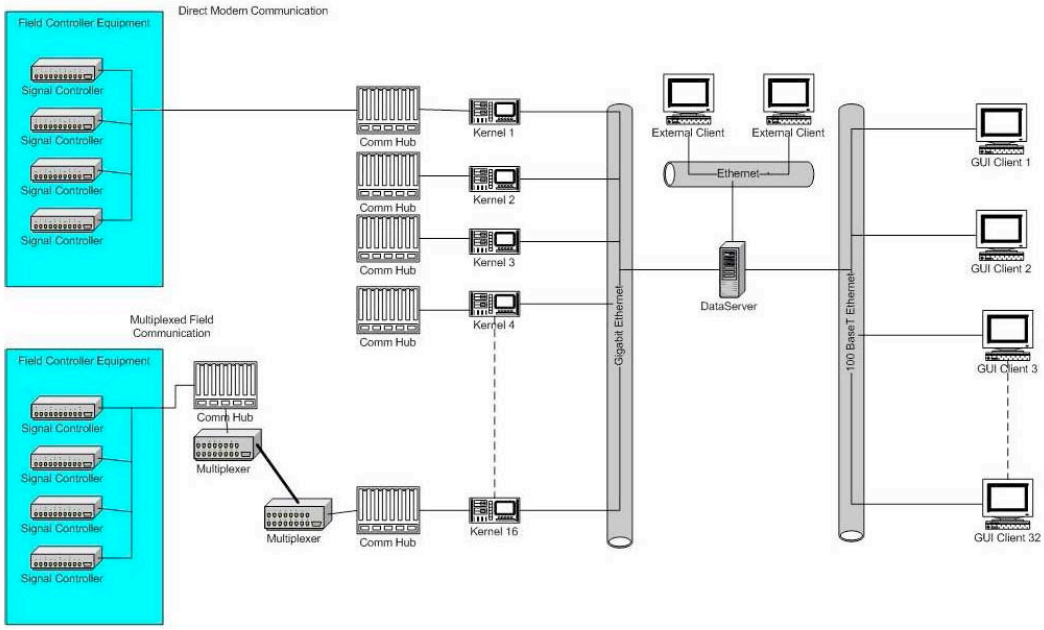


Figure 11: The LADOT Adaptive Traffic Control System Architecture

The complexity of developing such a system grows with the number of controlled intersections. While some computation can be carried concurrently (and they should, in order to speed up computation and meet real-time constraints) some others need to be synchronized because of data dependencies. The scenario is prone to error and it is easy to end up with data inconsistencies or system deadlocks. Because of changes in the traffic network it is often necessary to upgrade or modify some subsystems. The changes should be handled locally because a system shutdown is costly and not acceptable even during night hours. The problem grows with the size of the system. An urban grid often includes hundreds of instrumented intersections. The city of Los Angeles currently coordinates more that 1,300 intersections as a single system ((35)). The LADOT controls this system in a centralized manner, as described in figure 11. Every second the controllers are polled to retrieve the data collected by their sensors. The data is then aggregated in the *Data Center*. Various interdependent algorithms are then run to compute the optimal cycle length, splits and signal grouping based upon this data and new timing plans, if necessary, are uploaded on the controllers. The system is hierarchically

organized: at the leaf level the controllers are connected through serial ports to *communication hubs*. These hubs are connected to "*Kernels*", windows NT machine, through twisted pairs. A time division protocol ensures that all the necessary data exchanges take place without any delay each second. The *kernel* machines are then connected to the *Data Center* through a fiber optic network. The system was designed so that all the data from the sensors at the intersection and all the traffic control parameters for the controllers can be communicated without any delay. An unwanted consequence of the design is the difficulty to upgrade it. For example recently LA DOT upgraded the ATCS system to provide traffic priorities, i.e. trying to give on-demand green lights to delayed busses. The new Traffic Priority System (TPS) operates autonomously from ATCS, taking over control when necessary ((35)). TPS is a distributed system: it does not leverage on the existing communication and centralized computing infrastructure. It was more economical to equip the rapid bus corridors with a new communication system and leverage on the existing 2070s controllers for the computation than to use the ATCS infrastructure.

The tools in this paper would open the way to distributed implementation for ATCS-like systems and ease the development of TPS-like systems, where the control computes at the intersection controller and the coordination data flows through channels between intersections.

Using our tools, the traffic engineer could rely on Simulink to develop, simulate and tune the performance of their algorithms. Then they can obtain an implementation of their system directly using real Time Workshop, avoiding the cost and delay associated with going back to the vendor for re-programming. The signal would be controlled, to quote (20), "exactly the way the designer thinks it should be controlled".

5.2 Case study: Offset controller for Coordinated Traffic Signal

In this section an off-set controller for coordinated traffic lights along an arterial is developed. The traffic network of interest is described in 12. A major high traffic street is intersected by 4 minor low traffic roads. The addressed scenario is a peak hour asymmetric scenario, where almost all the traffic flow is in one direction of the major street with almost negligible turns and side street traffic. The intersection spacing is between 0.3 and 0.5 miles. In this scenario, as shown in (46), the total delay experienced by the vehicles is minimized using signal coordination. The idea is to create green waves on the main road so that a car that just got the right-of-way at the first intersection will get a green at all the intersections (see (28) and (8)). First the cycle length is fixed. Then the controllers are synchronized and their green phase are offset by $d * v$, where d is the distance between the two intersections and v is the target traffic speed. In all the pre-timed approaches as soon as the traffic speed deviates from the design speed the performance worsens. A possible solution to the problem was proposed in (1). His approach follows the actuated paradigm, where the offset is dynamically adjusted to reflect the real traffic scenario. At each cycle the offset is computed as before, but v is now the average vehicle speed measured in real-time.

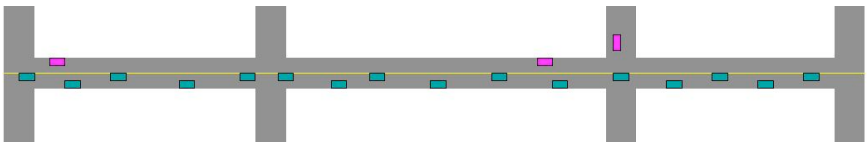


Figure 12: Asymmetric peak hour traffic on a major road intersected by four minor streets

The system has been implemented using Simulink, as in figure 13. The average speed has been computed using the Lighthill and Whitnam theory of traffic flow adding a white noise factor. 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.

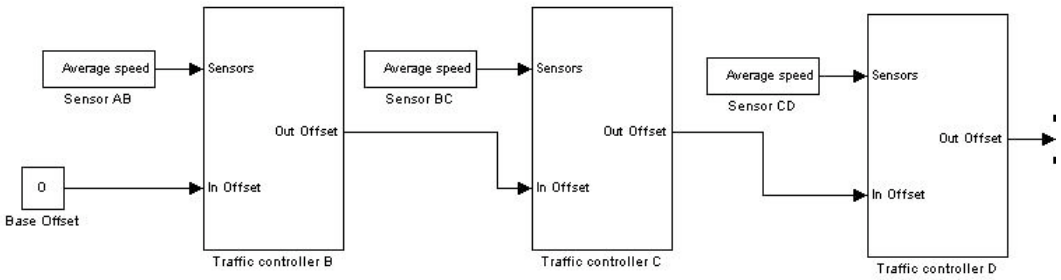


Figure 13: Simulink Model for the distributed traffic controller

The system behavior has been tested in the Simulink environment. In the test scenario an accident is happening between the first and the second intersection during the 100^{th} cycles and it is cleared out during the 180^{th}, and a minor one happened between the third and the fourth one during the 150^{th} cycle and it is cleared out during the 200^{th}. The offsets computed by the last three intersection (the offset for the first one is always 0) computed using the model in figure 13 are plotted in figure 14.

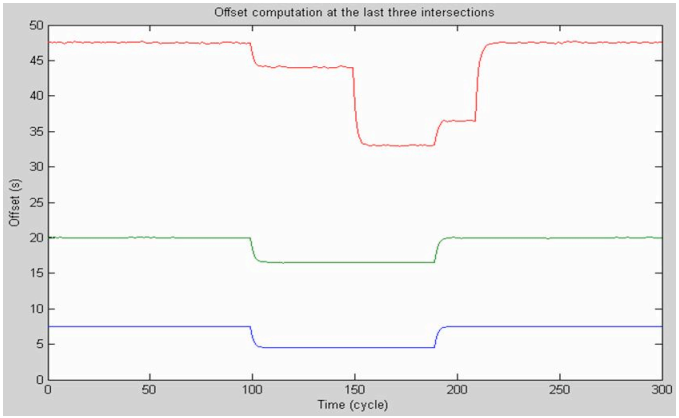


Figure 14: The offset as computed by the model described in figure 13

The test has been carried out on the same hardware used in the previous section. In this case performance has been measured as the total computation time needed to carry a step (i.e. from the end of the 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 switching 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 (20).

In the development of such a system it was not necessary to worry about synchronization and communication. The system is designed and implemented as if it was a centralized system. Then it was easily distributed using the BDSP library. The distributed system behavior is provably the same of the centralized one, it is less costly (there is no need for the central system, the computation can be carried over the existing controllers), and it can be design to degrade gracefully (because there is no single point of failure).

6 Conclusion and Future Work

New affordable, reliable and small sensor, communication and computing devices are enabling complex distributed systems, where components are updated frequently. Given the system size it is necessary to be able to handle local changes and updates locally, without impacting the overall system. For example, for the urban grid traffic controllers system described in section 5, if a new control strategy has to be implemented along an artery it should not be necessary to shut down the whole city network.

The research presented in this paper proposes a two step compilation scheme for distributed synchronous programming. The first step compiles the modular sequential code into modular semantically equivalent sequential asynchronous code. In this phase the code is annotated with node running time information and with the causality dependencies between its inputs and outputs.

Once the first step has taken place, the second step can be executed on the annotated compiled modules to ensure that they can be executed meeting the overall timing requirements and that they are not going to deadlock. If some modules are modified, only they need to be recompiled and then the second step can be carried over the old and new annotations to ensure that the modifications do not violate time constraints or introduce deadlock. Here the second step is developed only to be adequate for the signal control application. The resulting code is communicating over TCP/IP channels as fast as permitted by the input-output data dependencies. There is no global scheduler of computation or communication.

The first compilation step is modular structure preserving in the sense that any modification to a module of the synchronous program will only require recompilation of the altered module. In section 3 a compilation algorithm is presented and it is proved it preserves the semantic of the synchronous program in the sequential asynchronous compiled code. The main result then follows. The implementation is proved to be 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.

During this step, the 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.

The theoretical results are then transformed into software. The architecture of the BSDP library and its performance is presented. The results in this paper apply only to Simulink programs without causal loops (see section 2) used with discrete fixed-rate solver.

An application is explored in section 5, where an offset controller for an urban coordinated arterial deploying the Abu-Lebdeh speed-control algorithm is implemented in Simulink and compiled to execute distributed

over Pentium machines interconnected through TCP/IP channels.. Traffic control systems are increasing in complexity making them more costly to upgrade. The approach presented in this paper could make updates easier. The authors are currently working on implementing all the ATCS functionalities in the Los Angeles Adaptive Traffic Control Systems using the tools described in this paper.

References

[1] Abu-Lebdeh G., Benekohal R.F. "*Signal Coordination and Arterial Capacity in Oversaturated Conditions*", Journal of the Transportation Research Board, TRR 1727, TRB, National Research Council, Washington, DC, pp. 68-76.

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

[3] Benvenieste A., Caillaud B., Le Guernic P., *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.

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

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

[7] Boydstun M., "*Actuated Traffic Signal Systems*", National Institute for Advanced Transportation Technology, Traffic Signal Summer Workshop, 2004

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

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

[10] Caspi P., Ponzet M., "*Synchronous Kahn networks*", ACM. Sigplan Notices (Acm Special Interest Group on Programming Languages), vol.31, no.6, pp.226-38, June 1996.

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

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

[13] Cseh C. "*Architecture of the dedicated short-range communications (DSRC) protocol*, 48th IEEE Vehicular Technology Conference, VTC 98, Volume 3, 18-21 May 1998 Page(s):2095 - 2099 vol.3, 1998

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

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

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

[17] Francis J.G.F., The QR Transformation I, "*Comput. J.*", vol. 4, 1961, pp 265-271.

[18] Garcia A., Lucena C., Zambonelli F., Omicini A., Castro J., "*Software Engineering for large-scale multi-agent systems: research issues and practical applications*", Sringer Verlag, 2003

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

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

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

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

[23] Henzinger T. A., Kirsch C. M., Matic S., "*Schedule carrying code*". Proceedings of the Third International Conference on Embedded Software (EMSOFT), Lecture Notes in Computer Science, Springer-Verlag, 2003.

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

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

[26] Kahn G., "*The Semantics of a Simple Language for Parallel Programming*", Proceedings of the IFIP CongressŠ74. North Holland Publishing Company.

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

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

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

[30] Lee E. A., Sangiovanni-Vincentelli G. "*A Framework for comparing models of computation*", IEEE Transaction on Computer-Aided Design of Integrated Circuit and Systems, 17(12), 1217-1229, Dec 1998.

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

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

[33] Lee E. A., Zheng H., Zhou Y., "*Causality Interfaces and Compositional Causality Analysis*", Foundations of Interface Technologies (FIT), Satellite to CONCUR 2005, ENTCS TBD, San Francisco, California, USA, August 2005.

[34] Lee, Huang, Vaughn, Xiao, Hedrick, Zennaro, Sengupta, "*Startegies of Path-Planning for a UAV to Track a Ground Vehicle*", AINS Conference 2003.

[35] Li Y., Zhang W. B., "*Summary of Requirements: Los Angeles Transit Priority System*", California Patners for Advanced Transit and Highways, Insitute of Transportation Studies, University of California at Berkeley Internal report, 2005.

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

[37] Mak T., Xu P., Zennaro M., "*Wireless Communication Enabling Technologies for Intersection Decision Support System*", PATH Internal Report 2003

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

[39] Misra J., "*Distributed discrete-event simulation*", ACM Computing Surveys (CSUR), Volume 18 Issue 1 March 1986

[40] Rathinam S., Zennaro M., Mak T., Sengupta R., "*An architecture for UAV team control*", IAV Conference 2004, IFAC symposium on intelligent autonomous vehicles

[41] Ray J. C., "*Effective Traffic Signal Control*", Unpublished report, Sacramento County, California

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

[43] Ryan A., Zennaro M., Howell, Sengupta, Hedrick, "*An overview of emerging results in cooperative UAV control*", IEEE 2004 44th Conference on Decision and Control, December 2004

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

[45] Scheidler C., Heiner G., Sasse R., Fuchs E., Kopetz H. and Temple C., "*Time-Triggered Architecture (TTA)*", Advances in Information Technologies: The Business Challenge, IOS Press, ISBN 90 5199 385 4, 1997

[46] Skabardonis A., Bertini R.L. and Gallagher B. "*Development and Application of Control Strategies for Signals in Coordinated Systems*", Transportation Research Record 1634, Journal of the Transportation Research Board, Washington, D.C., 1998, pp. 110Ű117.

[47] Tanenbaum A. S., Van Steen M., "*Distributed Systems, Principles and Paradigms*", Prentice Hall 2002

[48] Van Olin, "*Advantages and disadvantages of traffic signals*", Institute of transportation and Traffic Engineering, University of California at Berkeley, Internal report UCB-ITS-RR-73-XX, 1973

[49] Jia Z., Varaiya P., Chen C., Petty K., Skabardonis A., "*Maximum throughput in LA freeways occurs at 60 mph*", version 4, IEEE 4^{th} international ITS conference, 2001

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

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

[52] Zennaro M., Sengupta R., "*Distributing Synchronous Programs Using Bounded Queues"*, 5th ACM International Conference on Embedded Software (EMSOFT'05), December 2005

[53] "*White Paper: UAV Over-the-Horizon Disaster Management Projects*", NASA-Ames Research Center, Moffett Field, California 2000

[54] "*Vehicle Infrastructure Integration (VII): Architecture and Functional Requirements*", version 1.1, Federal Highway Administration, ITS Joint program Office, July 2005.

[55] "*NTCIP 9001: National Transportation Communications for ITS protocol, guide*", American Association of State Highway and Transportation Officials, Institute of Transportation Engineers and National Electrical Manufacturers Association Information Report, version 3, 2002.

[56] "*Manual on Uniform Traffic Control Devices for street and highways*", Federal Highway Administration. Department of Transportation, Washington, D.C., 1970

[57] "*Traffic Manual, Chapter IX, Traffic Control Signals*", California Division of Highways, 1971

[58] "*Sepac Actuated Signal Control Software with NTCIP User manual*", Eagle SIEMENS, 2001

[59] "*Learning Simulink 5*", MathWorks edition, 2002

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

[61] Wimax Forum Web-Page
(http://www.wimaxforum.org/home)

[62] Dedicated Short range Communication Web-Page
(http://www.leearmstrong.com/DSRC/DSRCHomeset.htm)

[63] Berkeley Group Dedicated Short range Communication Web-Page
(http://dsrc.zennaro.net)

[64] California Center for Innovative Technologies Changeable Message Signs System Web-Page
(http://www.calccit.org/news/cms.html)

[65] SCOOT - Split Cycle Offset Optimization Technique Web-Page
http://www.scoot-utc.com

[66] Vehicle-Infrastructure Integration Initiative Web-Page
(http://www.its.dot.gov/initiatives/initiative9.htm)

[67] Intelligent Vehicle Initiative Initiative Web-Page
(http://www.its.dot.gov/ivi/ivi.htm)

[68] J. Paniati, U.S. Department of Transportation, talk at the National Transportation Operation Coalition
(http://www.ntoctalks.com/icdn/vii_pubmtg_v1.php)