Semantics-Preserving Implementation of Synchronous Specifications Over Dynamic TDMA Distributed Architectures

Dumitru Potop-Butucaru, Akramul Azim, Sebastian Fischmeister

Semantics-Preserving Implementation of Synchronous Specifications Over Dynamic TDMA Distributed Architectures

Dumitru Potop-Butucaru
INRIA,
Unité de recherche de Rocquencourt,
France
dumitru.potop@inria.fr

Akramul Azim
Department of Electrical and Computer Engineering
University of Waterloo,
Canada
aazim@uwaterloo.ca

Sebastian Fischmeister
Department of Electrical and Computer Engineering
University of Waterloo,
Canada
sfischme@uwaterloo.ca

ABSTRACT

We propose a technique to automatically synthesize programs and schedules for hard real-time distributed (embedded) systems from synchronous data-flow models. Our technique connects the SynDEx scheduling tool and the Network Code toolchain in a seamless flow of automatic model transformations that go all the way from specification to implementation.

Our contribution is the non-trivial connection between the models manipulated by SynDEx and by the Network Code toolchain, at both formal and tool level. We provide an algorithm for converting the data-dependent schedule tables output by SynDEx into Network Code programs which can be seen as an "assembly code" level for time-driven distributed real-time systems. The main difficulty is to ensure the preservation of both functionality and the real-time guarantees computed by SynDEx in the presence of clock drifts (which are abstracted away in the scheduling model of SynDEx). Existing tools can convert the resulting Network Code programs into software and hardware-accelerated execution units.

Categories and Subject Descriptors: D.3.4 [Programming languages]: Processors–*Code generation*; D.4.7 [Operating systems]: Organization and Design–*Distributed systems, Real-time systems and Embedded systems*

General Terms: Algorithms

Keywords: synchronous model, distributed real-time implementation, SynDEx, Clocked Graphs, Network Code, dynamic TDMA, clock synchronization

1. INTRODUCTION

This work addresses the implementation of real-time embedded control systems. In the context of model-driven development, software developers specify functions in a high-level data-flow language such as SCADE [20] or Simulink [8]. These formalisms follow cycle-based execution models, where the various *dataflow blocks* are cyclically executed in an order compatible with the *data dependencies* specified by *dataflow arcs*. *Conditional execution* mechanisms encode *execution modes* where each block is executed in specific *states* and for specific *inputs* of the system.

The cyclic execution model is also that of periodic real-time systems. However, producing efficient real-time schedules and implementations for data-flow specifications with data dependencies and conditional execution remains challenging. Spreading the application across a network of interconnected processors further complicates this problem, as algorithms need to consider several computing elements and communication lines.

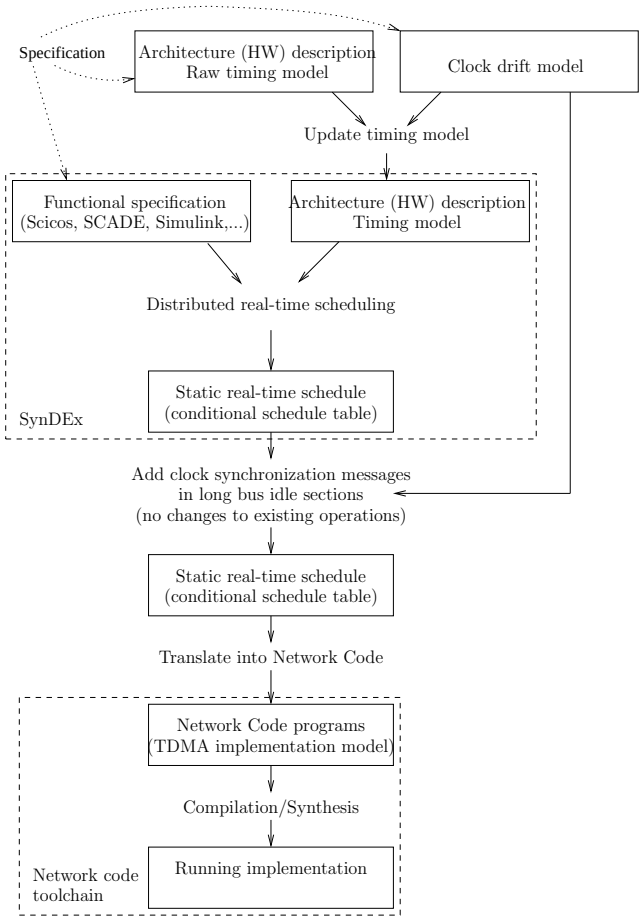
To allow formal reasoning of tractable complexity, related approaches [17, 23, 11, 29, 8] usually work with high-level abstractions of the execution hardware. For instance, Wu *et al.* [11] assume that communications take zero time, whereas Caspi *et al.* [8] work on time-triggered architectures (TTA), which already offer high-level services such as a global time reference.

Of particular relevance to this work is the architecture description formalism of SynDEx [17, 23]. It offers abstract notions of processor and contention-free communication line, including an execution model and timing information. The SynDEx tool takes as input an architecture description and a SCADE-like synchronous dataflow specification. It produces a model of the scheduled system in the form of a data-dependent schedule table assigning a fixed start date to each (conditioned) computation and communication. Existing "back-end" code generators use this schedule table to generate *event-driven* implementation code by basically implementing the abstract architecture model over the actual hardware.

In this paper, our synthesis target is the hardware abstraction layer provided by the Network Code formalism [13], which we use as an "assembly code" level for *time-triggered* distributed embedded systems. To improve the timing predictability of a processor and its communication interfaces, the Network Code formalism forces each computation and communication to run in a fixed amount of time. Existing tools can convert such programs into efficient implementations in both software and hardware.

Previous work relied on introducing new abstractions or algorithms to map the high-level models to low level exe-

EMSOFT'10, October 24–29, 2010, Scottsdale, Arizona, USA.



1: Proposed implementation flow. The dashed boxes identify the existing SynDEx and Network Code scheduling/synthesis flows.

cution models. Unlike such approaches, we **integrate in a single flow the existing SynDEx scheduling tool and the code generation toolchain built around the Network Code formalism** as pictured in Figure 1. Thus, we allow the automatic synthesis of hard real-time distributed (embedded) systems with **dynamic TDMA** communication buses from SCADE-like synchronous data-flow formalisms. **Our contribution is the non-trivial glue connecting the two tools and the associated formal models.**

To implement the schedule table generated by SynDEx, we synthesize the Network Code programs of each processor. To ensure the correct synchronization between the time-triggered Network Code programs on the separate processors of the network, a form of global time is needed on top of the Network Code formalism and tools. Our main contribution for this is the definition of an efficient clock synchronization mechanism that takes advantage of the specific form of the SynDEx-generated schedules. We use the clock drift model to (1) automatically update the timing model taken as input by SynDEx so that the output scheduling table also considers the overheads due to clock drift, and (2) to automatically update the output of SynDEx to insert clock synchronization communications whenever the bus is idle for too long (but all operations scheduled by SynDEx are left unchanged). The resulting implementation has no media access conflicts, so that the temporal guarantees originally computed by SynDEx hold.

The remainder of the paper is structured as follows: Section 2 gives a brief overview of related work. Section 3 presents the SynDEx scheduling flow, insisting on the definition of the functional specification, architecture, and static schedule models. Section 4 presents the Network Code formalism, and Section 5 gives an overview of our implementation flow. Then, Section 6 explains how Network Code programs are generated from static schedules, and Section 7 explains how the clock drift is accounted for. We conclude in Section 8.

2. RELATED WORK

We already cited several approaches to the (distributed) real-time implementation of conditional data-flow specifications. Our work differs from them in two main points: (1) the generation of dynamic TDMA communication protocols, and (2) the reliance on architecture abstraction techniques, instead of defining new scheduling algorithms.

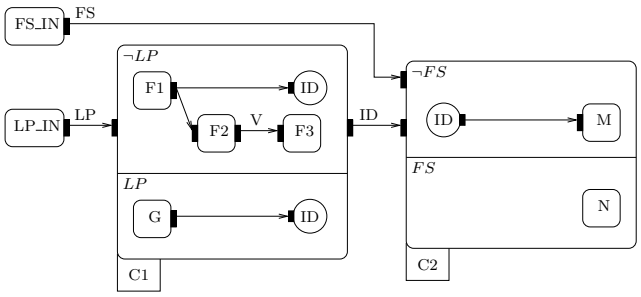
On the protocol level, our approach differs from existing work, as we automatically synthesize an application-tailored optimized medium access protocol. Traditional real-time and embedded networking protocols grant applications only limited control over the communication behaviour at run time. For example, all message identifiers (=priorities) on a CAN bus [24] must be unique and application developers usually ensure this by statically assigning priorities to messages and thereby defining the behaviour offline. Although it is technically feasible to reserving multiple identifiers for the same messages, this approach complicates the design. More flexible protocols for embedded networking like Powerlink Ethernet [12], FlexRay [15], VARAN [27], FTT-CAN [1] and its derivate FTT-Ethernet [21] partially depend on application state information as for example an application can request to use the asynchronous slot at the end of cycle. Yet still, decisions in these protocols are made at the start of the communication round or earlier. In contrast to this work, our work tries to optimally tailor the communication and computation schedule to the needs of the data flow application by permitting decisions at anytime in the schedule.

Tailoring the communication and computation behaviour to data-flow dependencies can significantly improve the performance. Several case studies across different application areas showed this including control theory [28], hybrid systems [4], video-on-demand, hierarchical scheduling frameworks [10] and in general bursty demand models [9, 22].

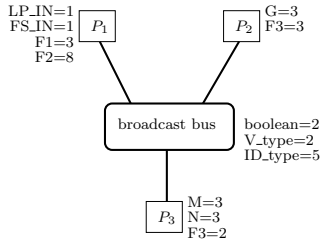
Some recent work explores similar ideas, but use different mechanisms. For example in [28], the authors generate so-called *state-based schedules* from high-level specifications such as control systems. Their work as well as the initial work [2] uses automata to express the schedule and thus requires regular specifications. Our work uses the notion of communication rounds which permits a lower complexity for analysis, verification, and generation [3].

3. THE SYNDEX SCHEDULING APPROACH

The AAA methodology and the SynDEx scheduling tool [17, 18] have been developed by a team lead by Yves Sorel to allow the fast automatic generation of efficient distributed real-time implementations of synchronous dataflow specifications. As pictured in Figure 1 (the upper dashed box), the



2: Example of dataflow specification



3: Example of architecture specification

SynDEx scheduling tool takes as input a functional specification and a timed model of the target execution architecture, and produces a schedule table which is a model of the statically-scheduled real-time implementation. To exemplify the SynDEx flow we shall use throughout the paper a simple example taken from [23].

3.1 Functional specification

The functional specification formalism of SynDEx is a hierarchical synchronous dataflow language similar to Lustre [20] or its SCADE graphical counterpart. However, various gateways allow the use of specifications written in Scicos [6] (a free Simulink-like formalism), Signal [19] (a synchronous dataflow language), etc.

A SynDEx specification consists of dataflow blocks, which have input and output ports, and dataflow arcs. Each dataflow arc connects one output port to one input port. At each execution cycle where it is executed, a dataflow block reads all its input ports and computes all its output ports. Dataflow nodes can be *elementary*, which corresponds to calls to library functions performing elementary computations such as "`+`", "`read_data`" etc., or *composed nodes* defined as a dataflow formed of other nodes. The execution of a composed node takes place as if the node has been replaced by its dataflow *expansion*. To allow conditional execution, a composed node can have several expansions, each one with its own *activation condition* telling when the particular expansion is used. Such nodes are called *conditioned nodes*.

Figure 2 shows an example input for SynDEx. Instead of the standard graphical representation, we use an *ad-hoc* one allowing the compact representation of the hierarchy levels. The specification represents a system with two switches (Boolean inputs) controlling its execution: low precision (LP) vs. normal precision ($\neg LP$), and fail-safe (FS) vs. normal operation ($\neg FS$). In low-precision mode, less operations are executed than in the normal precision mode. In the fail-safe mode the actuation operation that gets executed (N) does not use any of the inputs, because the sensors or treatment chain are assumed to be faulty (control is done using default values).

The specified system behaves as follows: At each execution cycle, the dataflow nodes FS_IN and LP_IN read FS and LP from the environment. If $LP = false$ then the execution of the conditioned node $C1$ is given by its expansion of the upper tab (labeled with activation condition $\neg LP$). If $LP = true$, then the expansion of the lower tab is used, with activation condition LP. Similarly, the upper and lower tabs of $C2$ give its expansions with activation conditions FS, respectively $\neg FS$. The input and output ports of the conditioned nodes are represented with circles in the expansions (as opposed to boxes, which are reserved to nodes). Note that the ID output of $C1$ is computed by both of $C1$'s expansions, but only the upper expansion of $C2$ uses the it in computations. The activation conditions of the expansions of a conditioned node are Boolean expressions over the inputs of the node.

Dataflow blocks having no dependency between them can execute concurrently. For instance, if $FS = true$ then N can be executed as soon as FS is read, independently of the execution of $F1$, $F2$, $F3$, or G. On the contrary, the computation of M must wait until both FS (with value $false$) and ID have arrived.

3.2 Architecture description

A hardware architecture description in SynDEx is a bipartite graph defining the interconnect between processors (computing elements) and communication lines, and annotated with timing information. The execution model associated to processors and buses, as well as the associated timing model remains abstract, but we shall see that it is adapted to our TDMA target platforms. An example of architecture description is given in Figure 3. It has 3 processors, named P_1, P_2, and P_3, and one broadcast bus connecting them.[1]

Each processor is capable of executing one sequential program describing the execution of the assigned dataflow nodes. Each bus is capable of executing a sequence of non-overlapping broadcast message passing communications (all processors receive all messages sent on a bus). The buses are free from errors (permanent or transient). They do not provide contention detection/handling mechanisms, because SynDEx will ensure that no time frame is assigned to two communications that can happen in the same execution cycle (and successive execution cycles do not overlap).

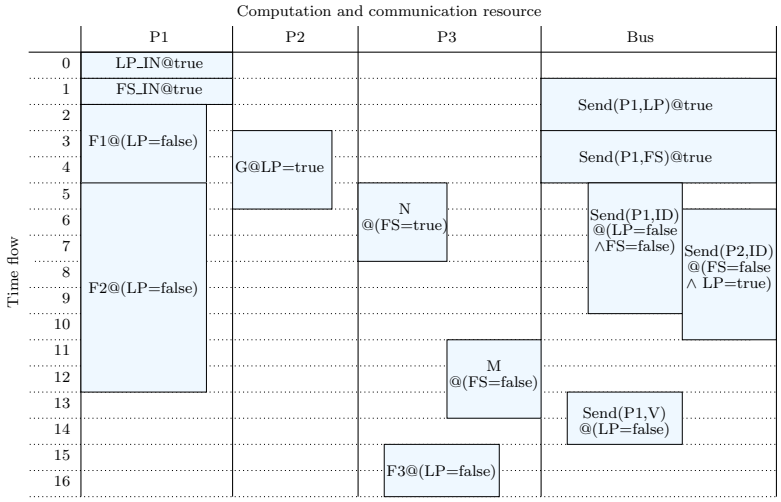
3.2.1 Timing model

To allow real-time scheduling, durations are associated to all:

- Elementary dataflow blocks on the processors that can execute them. For instance, node $F1$ can be executed only by P_1, and its duration is three time units.
- Data transfers of the various types of messages exchanged between nodes, on the buses that can perform these communications. For instance, the transmission of a data of type ID_type can be done by our bus in five time units.

In our case, most dataflow nodes can only be executed on a single processor, the notable exception being $F3$, which

[1]Other types of buses and communication lines can be defined, but we are not interested in them in this paper.



4: Schedule table for our example

can be executed, with different durations, by both P_2 and P_3. The durations can be used to specify complex partial allocations of the computations to processors, be them due to physical constraints (e.g., I/O operations on the I/O processor), or to allocation patterns desired by the user.

Real-time durations are interpreted as worst-case durations in the absence of all interference.

All control (tests, branching, protocol stacks, etc.) are assumed to be executed in zero time and their actual cost must be included in the worst-case durations provided above. The system has a precise global real-time reference and uses time-triggered execution of computation and communication based on this global clock.

3.3 Static scheduling

Recall that our synchronous functional specifications have a cycle-based execution model where the same *finite* description of decisions and computations is traversed at each execution cycle. Then, a natural way of of providing a real-time schedule of its *infinite* execution is to compute a schedule of one execution cycle. This finite schedule is then repeated each repetition representing one cycle.

This is the approach taken by SynDEx. The schedule computed for our small example is given in Figure 4. The schedule is static, in the sense that each computation and communication has a fixed start date inside the execution of the cycle.

Figure 4 shows the schedule table. Each column represents one processor or the bus. The description of each *scheduled operation* o (computation or communication) includes the operation to be performed (denoted $operation(o)$), the start date $date(o)$, the duration $duration(o)$, and its activation condition $clk(o)$. For instance, the last operation on the bus is the operation `send` by processor $P1$ of the variable V. This operation takes two time units and is executed in cycles where $LP = false$.

At most one operation is active at a time on each processor or bus. Therefore, two operations in one column can only overlap if their activation conditions are mutually exclusive. In our example, the two communication operations that overlap in time have to have exclusive activation conditions. In Figure 4, the width (horizontal projection) of the operation boxes intuitively represents their activation conditions, in a way similar to Venn diagrams. Two operations have exclusive conditions when their projections are exclusive.

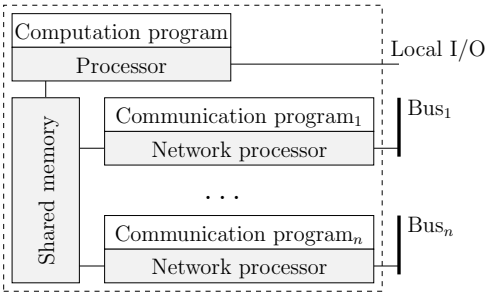
The scheduling algorithm of SynDEx respects the data dependencies of the specification. More precisely, in the schedule table, all data needed to perform a scheduled operation o is available at the operation start. When o is a computation on a processor, all the variables needed to compute the activation condition $clk(o)$ and all the inputs of the dataflow node $operation(o)$ are available on the processor at date $date(o)$. When o is the sending of variable V by processor P, then (1) V is available on P at date $date(o)$, and (2) all the variables needed to compute $clk(o)$ are available on all processors connected to the bus at date $date(o)$. Condition (2) ensures that all processors can decode all bus traffic.

These properties ensure that the bus schedule is easily implementable as a set of distributed programs on the actual hardware. Various back-ends [17] exist translating such schedule tables into code running over asynchronous communication buses such as CAN [24] or Ethernet. We shall see in the next section that the time-triggered nature of these tables is easily mapped onto the time-triggered execution model of Network Code.

4. THE NETWORK CODE FORMALISM

Network Code is a domain-specific programming language for the implementation of distributed real-time systems with time-triggered (TDMA) communication systems. It originates from the goal to functionally and temporally tailor properties of the computation and communication behaviour. The formalism specifically concentrates on messaging and the media access control layer. The language consists of a small set of assembly-like instructions with well defined operational semantics. Besides software prototypes, Network Code has been implemented as a hardware-accelerated special processor [14] on top of Ethernet and inside a network switch [7]. We use Network Code to give a time-triggered implementation of the static schedule table generated by SynDEx.

Network Code assumes a set of *computation units*, corresponding to the processors of a SynDEx architecture de-



5: Internal organization and execution model of one processor

scription, and a set of broadcast buses connecting these processors.

The buses use a time-division multiple access (TDMA) communication scheme which, if correctly scheduled, eliminates all errors due to message collision/interference. Communication slots and rounds form the building blocks of such a scheme. A slot starts at a given time and ends after a known duration. Within the slot's time span at most one computation unit has write access to the bus. A sequence of slots defines a communication round after which the behaviour repeats.

```
1  START: wait(1)
       L1: if true then
3            future (L2,2)
             send (bus_id, sizeof(LP), LP)
5            halt ()
           endif
7          wait (16)
           goto (START)
9      L2: if true then
             future (L3,2)
11           send (bus_id, sizeof(FS), FS)
             halt ()
13         endif
           wait (14)
15         goto (START)
       L3: if (not LP) and (not FS) then
17           future (L5,8)
             send (bus_id, sizeof(ID), ID)
19           halt()
           endif
21         wait (1)
           goto (L4)
23     L4: if LP and (not FS) then
             future (START,11)
25           wait (5)
             receive (bus_id,ID)
27           halt()
           endif
29         wait (7)
           goto (L5)
31     L5: if (not LP) then
             future (START,4)
33           send (bus_id, sizeof(V), V)
             halt()
35         endif
           wait (4)
37         goto (START)
```

Listing 1: The Network Code program generated from the schedule table of Figure 4 for the network processor of P_1

Figure 5 outlines the architecture of one computation unit in our system showing the hardware in greyed boxes and the software in white boxes. To execute the computation operations (in our case dataflow nodes) scheduled on it, a computation unit has one *computation processor*. To control the TDMA bus connections (i.e., send and receive messages), a computation unit has one *network processor* per connected bus. The processor and network processors are connected through a shared memory. They also share a common time reference, called the clock of the computation unit. Each of the computation and network processors is executing exactly one program and the programs of a computation node are in sync due to the common time reference.

In our concrete implementation, the network processor is an application-specific instruction set (ASIP) processor. The Network Code toolchain (the lower dashed box in Figure 1) takes the Network Code programs associated to the network processors of the various computation units and generates a binary executable for the available hardware-accelerated implementation called the Network Code Processor [14]. The binary executable contains the Network Code program and the memory setup as a byte stream that is uploaded into the configuration memory of the FPGA hardware. It also takes the Network Code program of the computation processor and generates executable code.

Listing 1 shows a simplified, readable version of the the Network Code program which we automatically synthesize from the schedule table of Figure 4 for the network processor of the P_1 processor of our example. This program handles the bus communications of P_1 and implements the bus schedule of Figure 4. It starts by waiting for two time units. Then, it sends on the bus a message containing the value of LP. The `future` statement of line 3 sets up a timer that will jump to label `L2` after two time units. The combination of `future` and the `halt` statement in line 5 (which blocks execution) ensures that `L2` will be reached at date four, even if the `send` completes earlier.

Starting from label `L4`, the program shows the code corresponding to the send of ID by processor P_2 (which P_1 receives). The execution condition of the receive operation ($LP \wedge \neg FS$) guards its execution. The `receive` statement simply collects the value of ID from a buffer after the completion of the actual communication (after waiting for 5 time units). The `future` statement prescribes a jump to the program start, because after the receive operation completes there is no bus operation of compatible activation condition in the current execution cycle. Therefore, we need to start the next cycle (after the prescribed ten time units, counted from the beginning of the receive operation).

5. FRAMEWORK OVERVIEW

The main advantage in giving a Network Code-based implementation to the SynDEx-generated schedule tables is that both formalisms have a time-triggered execution model. However, several problems remain.

The main formal problem is that the Network Code formalism does not define a global time reference for the distributed platform. Therefore, the development of a running TDMA distributed system using Network Code necessarily passes through the definition of a clock synchronization mechanism. Moreover, the use of SynDEx for scheduling means that the clock synchronization overheads must be compatible with the timing model of SynDEx. We shall address this problem in Section 7. **Until then, we shall assume that we have a global time reference.**

To complete the definition of our implementation problem,

we assume that (1) there are no permanent or transient message transmission errors on the bus, and (2) there is no need for data packetization, the data being already cut into small-enough pieces at specification level. These assumptions will be the subject of future work.

Under these assumptions, our implementation problem is that of synthesizing the Network Code programs implementing the schedule table of SynDEx. For each processor P specified in the SynDEx architecture description (=Network Code computation unit), we need to synthesize:

- One "computation program", denoted $ComputationProgram(P)$
- One "communication program", denoted $NetworkProgram(B, P)$ per bus B the processor P is connected to (the program of the associated network processor).

It is interesting to note that existing SynDEx back-ends structure the generated event-driven code in a similar way, with one *computation thread* and one *communication thread* per connected bus for each processor.

The main difficulty during the generation of the Network Code programs is related to the move from the absolute time references (start dates) of the SynDEx schedule tables to the relative time (timeouts) manipulated by the `future` and `wait` statements of Network Code.

6. NETWORK CODE GENERATION

This section explains how the Network Code programs mentioned above are generated from the schedule table. We provide the two algorithms generating respectively: (1) the Network Code programs of the various bus interfaces and (2) the computation program.

6.1 Network Processor code generation

To simplify the algorithm, we introduce the following definitions and notations:

- λ denotes the global length of the schedule.
- $\mathcal{S}_B$ is the set of communication operations scheduled on bus B.
- $D = \{d \mid \exists o \in \mathcal{S} : date(o) = d)\}$ is the set of dates of operations in $\mathcal{S}_B$. We shall assume that $D = \{d_1, \ldots, d_n\}$ with $d_1 < \ldots < d_n$.
- Under the previous notation, the generated network code program uses $n+1$ jump labels, which are `START`, `L1`, ..., `L`n.
- $\mathcal{S}_B^i = \{o \in \mathcal{S}_B \mid date(o) = d_i\}$ is the set of bus operations scheduled on date d_i for some $1 \leq i \leq n$. We shall assume that $\mathcal{S}_B^i = \{o_1^i, \ldots, o_{k^i}^i\}$ where $k^i > 0$ is the number of elements of $\mathcal{S}_B^i$.
- $clk^i = \bigvee_{j=1}^{k^i} clk(o_j^i)$ is the union of the activation conditions of the operations of $\mathcal{S}_B^i$.
- c^i is the activation condition defining when `L`i is jumped to. It is computed by the translation algorithm. By construction, clk^i implies c^i.

The algorithm building the Network Code program $NetworkProgram(B, P)$ is provided in Function 1. It follows the simplified Network Code conventions used throughout the paper for the `send` and `receive` primitives:

- The `send` operations take as argument the bus identifier, the variable (memory zone), and the variable size.
- The receive operations take as argument a variable (memory zone) which must be large enough to allow the storage of the transmitted data.

Function 1 Computation of $NetworkProgram(B, P)$

Input: P, $\mathcal{S}_B$, `bus_id`, λ
Output: $NetworkProgram(B, P)$

for $i = 1$ **to** n **do** $c^i := false$ **done**
for $i = 1$ to n **do**
  for $j = 1$ to k^i **do**
    {**Step 1:** Build the code for o_j^i}
    if $operation(o_j^i) = Send(P, V)$ for some variable V **then**
      Let OP_j^i be the piece of code:
        `send (bus_id, sizeof(V), V)`
    else
      {$operation(o_j^i) = Send(P', V)$ for some $P' \neq P$}
      Let OP_j^i be the piece of code:
        `wait(`$duration(o_j^i)$`)`
        `receive (bus_id,V)`
    end if
    {**Step 2:** Build the `future` statement}
    Let d_m be the smallest element of D with $d_m \geq d_i + duration(o_j^i)$ and such that there exists $o' \in \mathcal{S}_B(d_m)$ with $clk(o_j^i) \wedge clk(o') \neq false$.
    if such a d_m exists **then**
      $c^m := c^m \vee clk(o_j^i)$
      Let F_j^i be the piece of code:
        `future(`$d_m - d_i$`,L`m`)`
    else
      Let F_j^i be the piece of code:
        `future(`$\lambda - d_i$`,START)`
    end if
    {**Step 3:** Assemble the full code for o_j^i}
    Let C_j^i be the piece of code:
      `if(` $clk(o_j^i)$ `) then`
        F_j^i
        OP_j^i
        `halt()`
      `endif`
  end for
  {**Step 4:** Where to go if no o_j^i is executed}
  Let d_m be the smallest element of D with $d_m \geq d_i + \max\{duration(o_j^i) \mid 1 \leq j \leq k^i\}$ and such that there exists $o' \in \mathcal{S}_B(d_m)$ with $clk(o') \wedge (c^i \wedge \neg clk^i) \neq false$.
  if such a d_m exists **then**
    $c^m := c^m \vee (c^i \wedge \neg clk^i)$
    Let F^i be the piece of code:
      `wait (`$d_m - d_i$`)`
      `goto (L`m`)`
  else
    Let F^i be the piece of code:
      `wait (`$\lambda - d_i$`)`
      `goto (START)`

43: **end if**
44: {**Step 5:** Assemble the full code for $\mathcal{S}_B^i$}
45: Let C^i be the piece of code:
46: Li: C_1^i
47: C_2^i
48: ...
49: $C_{k^i}^i$
50: F^i
51: **end for**
52: {**Step 6:** Final code assembly}
53: Let $NetworkProgram(B, P)$ be the piece of code:
54: `START: wait(`d_1`)`
55: C^1
56: ...
57: C^n

To clarify the way the algorithm works, we already gave in Listing 1 of Section 4 the Network Code program it generates from the schedule table of Figure 4 for the network processor of processor P_1. We also explained there the functioning of the generated code. We explain here how code generation takes place.

The algorithm works by grouping together all scheduled operations of $\mathcal{S}_B$ with the same start date, and assigns a jump label to every such date/group. In our example, each operation has a different start date, so we have 5 jump labels (`L1` to `L5`) plus the `START` label defining the entry point of the program, which is also reached at the beginning of every cycle. Each of the Li labels points to a sequence of guarded statements, followed by a delayed unconditional jump statement. Each of the guarded statements corresponds to an operation of $\mathcal{S}_B$ belonging to the group associated with Li. In the guarded statement associated to operation o, the guard is the activation condition $clk(o)$, and the statement includes the actual code for executing $operation(o)$ (a `send` or `receive` statement). The remaining `future` and `halt` statements ensure 2 important properties:

- The operation code takes a fixed amount of time (the one specified by the schedule).
- After this fixed duration, control is given to the next Li jump label (=date in the schedule table) where an operation may be activated. If no such jump labels exist, control is given to `START`.

For instance, the guarded code after label `L3` encodes the `Send(`$P1$`,ID)(LP=false`$\wedge$`FS=false)` operation of Figure 4. Starting from its start date (5), the next time date where an operation has an activation condition which is not exclusive with the one of our operation is 13. Therefore, the `future` statement is set to give control in 8 (=13-5) time units to the label `L5` corresponding to date 13.

It is possible that even if we jump to a certain Li label, none of the associated operations are activated. This can happen in cases where $c^i \wedge \neg clk^i \neq false$. In our example, `L3` exhibits this behavior, because $c^i = true$ and $clk^i = ((\texttt{LP} = false) \wedge (\texttt{FS} = false))$. To cover these cases, the code following each label ends with an unconditional jump to the next jump label where an operation has a condition which is not exclusive with $c^i \wedge \neg clk^i$. The computation of the c^i activation conditions is realized in lines 1,17, and 35 of the algorithm.[2]

[2]The advantage of this translation scheme is its simplicity and closeness to the scheduling tables of SynDEx, in the sense that execution decisions are done at the operation start date, not before. Better algorithms can be built which anticipate these decisions, minimizing at the same time the number of jumps. However, this requires a form of lifetime analysis which would needlessly complicate this paper.

The algorithm works as follows: Step 1 builds the code for the actual `send` or `receive` operations. Step 2 builds the associated `future` statement, and Step 3 assembles the whole guarded code. Step 4 builds the unconditional jump code, and Step 5 builds the whole code associated with a jump label Li. Finally, Step 6 assembles the whole program.

Note that the generated code includes the jumps to the `START` label that are needed to start a new computation at each cycle. Also note, in lines 14 and 32, the slight optimization that uses an analysis of the activation conditions to minimize the number of jumps between labels (the code would also work by replacing Lm with the label L$(i+1)$, but with larger computational requirements). Of course, the code can still be largely optimized.

6.2 The computation program

Given that the execution model is identical to that of the communication programs, the only modification needed in Function 1 to generate $ComputationProgram(P)$ is in Step 1, which must be fully replaced with:

1: {**Step 1:** Build the code for o_j^i}
2: Assume $operation(o_j^i)$ is the execution of function F
3: Let OP_j^i be the piece of code:
4: `call (F,`*`parameter_list`*`)`

where the `call` statement executes the library function F (provided by the user) with the given parameter list (list of variables).

7. CLOCK DRIFT MANAGEMENT

We have seen in the previous section that the Network Code language includes instructions for temporal control, such as `wait(d)` and `future(d,l)`. Given the nature of our time-triggered implementations, the system must measure time consistently on all processors. So far, we assumed that, for instance, any two calls of `wait(d)` (on the same processor, or on different processors) will wait for the same duration on a real-time clock. In this section, we investigate how we provide means to realize this assumption, so all processors precisely execute the scheduling table produced by SynDEx.

In general, if the assumption is false, then the generated code will be *a priori* incorrect. In the context of Figure 4, if the clock of P_2 is faster than that of P_1, then it will potentially trigger the execution of G (at local date 4) while the data needed to compute its activation test (LP) is still unavailable on P_2 (*i.e.*, not received and therefore potentially invalid). Similar problems occur if the speeds of the local clocks can change over time.

The assumptions made in Section 5 require us to address the classical problem of clock synchronization in distributed systems. A number of different approaches exist for this problem [25, 5, 16, 25, 26, 30]. Our novel contribution to this area is that we fit the clock drift management around the application demands. Previous work treated clock drift as a separate problem, different from the application. We

integrate the clock drift management directly into the application and the communication schedule.

Our approach has two main elements: (1) Take advantage of our specific scheduling approach to integrate a clock drift model directly in our scheduling and synthesis flow and (2) if necessary, use an additional standalone clock synchronization algorithm on top of the integrated drift model.

During the synthesis, we have full control over the computations and communications in the system. Thus, our tool can exert fine control over the communication structure to attain very low synchronization overhead without changing the already scheduled computations and communications generated by SynDEx. Supplementary synchronization messages will only be added, if the bus is idle for long periods of time, to ensure that clock drift remains within the required bounds. The advantage of our technique is that the SynDEx scheduling tool can be used *as-is*, without changes to the scheduling policies to account for clock drift gradually along the schedule.

For simplicity, we define our clock drift management technique for the case where the system has a single bus. To define it, we assume that the developer knows that the target hardware satisfies the following properties:

1. The real-time durations of any two `wait(d)` statements in the system differ by less than $\alpha * d$, where α is a given constant.

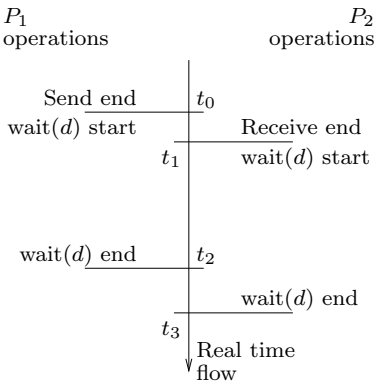
2. The low-level communication hardware (in our implementation, Ethernet controllers) detects and signals the **end** of `send` and `receive` operations. These events serve as clock synchronization points. Moreover, the end of a `receive` operation occurs after the end of the corresponding `send`, but less than β time units later, where β is a given constant.

3. The duration of a communication can be precisely computed from the length l of the transmitted data using a function $comm(l)$.

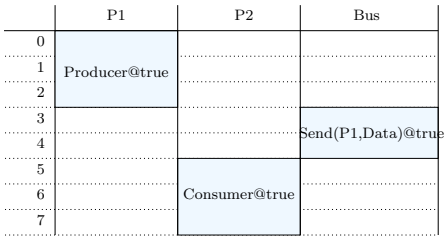
In addition, we assume that the developer provided a full SynDEx architecture description, including timing information. We shall denote with γ the longest duration of a bus communication specified in the architecture description.

Recall now that the SynDEx scheduler ensures that all processors know the expected start date of each communication. Assume that a communication operation o with start date s_o transported data of legth l_o. Then, the expected date of the operation end is $e_o = s_o + comm(l_o)$. Then, using the communication end events to change the local clock value of all processors to e_o will synchronize them with an accuracy of β. The timers associated with `future` and `wait` statements must be updated with the difference between the old clock value and e_o. If the timers reach 0 through this operation, the associated jumps are triggered. **From now on, we shall assume that this clock synchronization mechanism is provided by the Network Code code generators over the given hardware.**

However, by itself, this clock synchronization mechanism does not ensure the correctness of the implementation because the clock synchronization is not exact and because between the synchronization points the clocks may drift further apart. Assume, for instance, that $\beta = 0$, $\alpha = 0$, and that the duration of a `send` operation is exactly that specified in the SynDEx schedule. Then, if the receiver has a faster clock than the sender, then the `wait` statement of the receiver will trigger the reading of the reception buffer before the send is complete, potentially resulting in corruption of data.



6: Clock drift after a bus communication. Here, we have $-\alpha * d \leq t_3 - t_2 \leq \beta + \alpha * d$. No supplementary communication or piggybacking of time information is needed to achieve this precision.

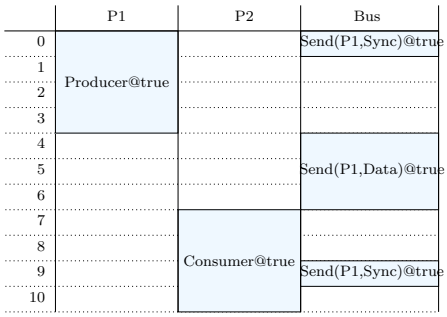


7: Scheduling table for a producer-consumer example on two processors

Figure 6 shows the model for computing the clock drifts starting from a clock synchronization point (communication end). If we bound the bus idle space between two successive communications, then we obtain a bound on the clock drift. For instance, if we bound the idle space by γ, the real-time distance between the timeouts of two `wait` or `future` statements measured on two different processors and starting from the same synchronization point is bounded by $-2 * \alpha * \gamma \leq t_3 - t_2 \leq \beta + 2 * \alpha * \gamma$ (because the d of Figure 6 is bounded by $2 * \gamma$).

When applying this drift model to the schedules generated by SynDEx and implemented into Network Code, the first problem that arises is that a processor with a faster clock can terminate a `receive` while the send is incomplete (the case where $t_3 - t_2$ is negative). To ensure that this never happens, we have to reserve more time than actually needed by the message transmission to compensate for the differences in clock speed and bus access overheads. Thus, to schedule a message of type t (with transmission duration $duration(t) = d_t$), SynDEx will use instead a longer duration d'_t to ensure that the reception is never interrupted. Under the given drift model, the best (smallest) value for d'_t to ensure correctness is $d_t + \lceil 2 * \alpha * \gamma \rceil$. A similar problem occurs with computations, meaning that we need to add $\lceil 2 * \alpha * \gamma \rceil$ to each operation duration. The updated timing model is the one given as input to SynDEx.

Finally, the output of SynDEx needs to be updated to ensure that the bus cannot be idle for longer than γ time



8: The example of Figure 7 with clock drift management

units. To ensure this, clock synchronization communications are inserted in bus idle sections, before giving the resulting scheduling table to our translation algorithms. When no specific synchronization mechanisms exist on the bus, clock synchronization messages can be assumed to be communications of the shortest available type the bus can transmit (so that the communication takes minimal time on the bus).

The generated code ensures exclusive bus usage by itself, thus the system is free of collisions and requires no white space detection mechanism.

We show how our clock drift management technique works using two examples. The first one is a very simple producer-consumer example. Its schedule table generated by SynDEx without the use of clock drift management is given in Figure 7. The producer on P_1 executes the code to generate the data. Then it communicates the data on the bus, and finally the consumer in P_2 executes and uses the data. To produce the schedule table using our clock drift management technique, we assume that $\lceil 2 * \alpha * \gamma \rceil = 1$, so that the duration of each computation and communication is increased by 1 in the timing model given to SynDEx. The output of our technique is given in Figure 8. This schedule table includes the dataflow operations scheduled by SynDEx, as well as the two **Sync** operations added in the long bus idle sections. We have assumed that the shortest bus communication takes 1 time unit.

The second example, given in Figure 9 shows the schedule table of our initial example (of Section 3), as produced when using the clock drift management. We assumed that $\lceil 2*\alpha*\gamma \rceil = 1$ and that the **Sync** operations take the shortest duration of a communication, as specified by the architecture description of Figure 3 (no increment is needed, because we are not interested in the data). A single, conditioned **Sync** operation is added, represented by the two boxes with identical label in our graphical representation.

8. CONCLUSION

Model-driven development relies on sound mechanisms and tools for generating code from high-level specifications. A well understood approach is to create abstractions for modelling, architecture, and executing systems and rely on their semantics when generating code. A number of abstractions exist and related work defines usually one and then relies on other work to combine the different models.

In this work, we create a full suite as a framework in which we combine a modelling abstraction (synchronous data-flow), an architecture abstraction (execution, communication durations, and clock drift), with an execution abstraction (time-triggered computation and execution as found in the Network Code formalism). We show how we solved the challenges occurring when building such a framework and the accompanying algorithms, and we provided a guiding example to illustrate the tool chain.

For the future, we mention here only two extensions of our implementation flow which seem particularly interesting. The first one is the ability to take as input multi-clock synchronous specifications, and to output multi-period real-time implementations. This implies extensions to both the SynDEx flow, and the SynDEx-Network Code glue.

The second extension line aims at refining the execution model of the Network Code formalism, to take into account the costs of control. The current approach, which considers that all tests, jumps, and timeouts take no time works well when these costs are negligible with respect to the costs of communications and dataflow computations. But having a finer accounting of control costs would allow us to handle specifications of finer grain. This direction should also include work on (1) the synthesis of time-triggered code that minimizes the cost of control, and (2) better clock synchronization mechanisms.

9. ACKNOWLEDGEMENTS

This research was supported in part by NSERC DG 357121-2008, ORF RE03-045, and ISOP IS09-06-037.

10. REFERENCES

[1] L Almeida, P. Pedreiras, and J.A.G. Fonseca. The FTT-CAN protocol: Why and how. *IEEE Trans. on Industrial Electronics (TIE)*, 49(6):1189–1201, December 2002.

[2] R. Alur and G. Weiss. Regular specifications of resource requirements for embedded control software. In *Proceedings RTAS'08*, Washington, DC, USA, 2008.

[3] M. Anand. *Conditional models for compositional design of real-time embedded systems*. Ph.D. thesis, University of Pennsylvania, Philadelphia, PA, USA, May 2008.

[4] M. Anand, S. Fischmeister, Y. Hur, J. Kim, and I. Lee. Generating reliable code from hybrid systems models. *IEEE Transactions on Computers*, 2010. To appear.

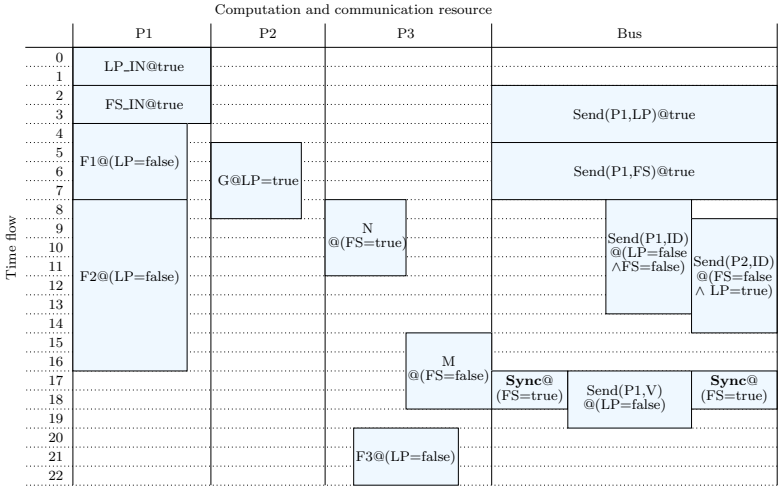
[5] E. Armengaud and A. Steininger. Remote measurement of local oscillator drifts in FlexRay networks. In *Proceedings DATE'09*, pages 1082–1087, 2009.

[6] S. L. Campbell, J.-P. Chancelier, and R. Nikoukhah. *Modeling and Simulation in Scilab/Scicos with ScicosLab 4.4 (second edition)*. Springer, 2010.

[7] G. Carvajal and S. Fischmeister. A TDMA Ethernet switch for dynamic real-time communication. In *Proceedings FCCM'10*, Charlotte, United States, May 2010.

[8] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From Simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications. In *Proceedings LCTES'03*, San Diego, California, USA, 2003.

[9] S. Chakraborty, L.T.X. Phan, and P.S. Thiagarajan. Event count automata: A state-based model for



9: Schedule table for our example, including clock drift management

stream processing systems. In *Proceedings RTSS'05*, Washington, DC, USA, 2005.

[10] A. Easwaran, M. Anand, and I. Lee. Compositional analysis framework using EDP resource models. In *Proceedings RTSS'07*, Washington, DC, USA, 2007.

[11] P. Eles, K. Kuchcinski, Z. Peng, A. Doboli, and P. Pop. Scheduling of conditional process graphs for the synthesis of embedded systems. In *Proceedings DATE'98*, Paris, France, 1998.

[12] Ethernet Powerlink Standadisation Group (EPSG). *Ethernet Powerlink V2.0 – Communication Profile Specification*, 2003.

[13] S. Fischmeister and I. Lee. A verifiable language for programming real time communication schedules. *IEEE Transactions on Computers*, 56(11):1505–1519, 2007.

[14] S. Fischmeister, R. Trausmuth, and I. Lee. Hardware acceleration for conditional state-based communication scheduling on real-time Ethernet. *IEEE Transactions on Industrial Informatics*, 5, 2009.

[15] FlexRay Consortium. *FlexRay Communications System – Protocol Specification*, June 2004. Version 2.0.

[16] M. Fugger, E. Armengaud, and A. Steininger. Safely stimulating the clock synchronization algorithm in time-triggered system: A combined formal and experimental approach. *IEEE Transactions on Industrial Informatics*, 5(2):132–146, 2009.

[17] T. Grandpierre, C. Lavarenne, and Y. Sorel. Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In *Proceedings CODES'99*, Rome, Italy, May 1999.

[18] T. Grandpierre and Y. Sorel. From algorithm and architecture specification to automatic generation of distributed real-time executives: A seamless flow of graphs transformations. In *Proceedings MEMOCODE'03*, Mont Saint-Michel, France, June 2003.

[19] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann. Polychrony for system design. *Journal for Circuits, Systems and Computers*, 12:261–304, 2002.

[20] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[21] P. Pedreiras, P. Gai, L. Almeida, and G.C. Buttazzo. FTT-Ethernet: a flexible real-time communication protocol that supports dynamic QoS management on Ethernet-based systems. *IEEE Transactions on Industrial Informatics*, 1(3):162–172, August 2005.

[22] L. T. X. Phan, S. Chakraborty, and P. S. Thiagarajan. A multi-mode real-time calculus. In *Proceedings RTSS'08*, Barcelona, Spain, 2008.

[23] D. Potop-Butucaru, R. Simone, Y. Sorel, and J. Talpin. Clock-driven distributed real-time implementation of endochronous synchronous programs. In *Proceedings EMSOFT'09*, Grenoble, France, 2009.

[24] Robert Bosch GmbH. *CAN Specification, Version 2*, September 1991.

[25] D. Salyers, A. Striegel, and C. Poellabauer. A light weight method for maintaining clock synchronization for networked systems. In *Proceedings ICCCN'08*, pages 522–526, 2008.

[26] K. Sun, P. Ning, and C. Wang. Secure and resilient clock synchronization in wireless sensor networks. *IEEE Journal on Selected Areas in Communications*, 24(2):395–408, 2006.

[27] VARAN–versatile automation random access network. `www.varan-bus.net`. Visited Mar. 2009.

[28] G. Weiss, S. Fischmeister, M. Anand, and R. Alur. Specification and analysis of network resource requirements of control systems. In *Proceedings HSCC'09*, San Fransisco, United States, April 2009.

[29] Dong Wu, B. M. Al-Hashimi, and P. Eles. Scheduling and mapping of conditional task graphs for the synthesis of low power embedded systems. In *Proceedings DATE'03*, Munich, Germany, 2003.

[30] M. Zhang, J. Shi S. Shen, and T. Zhang. Simple clock synchronization for distributed real-time systems. In *Proceedings ICIT'08*, pages 1–5, 2008.