Implementing Synchronous Reactive Components upon Multiprocessor Platforms

Sanjoy Baruah
Washington University in Saint Louis
baruah@wustl.edu

Abstract—Model-based design methodologies based on the synchrony assumption are widely used in many safety-critical application domains. The synchrony assumption asserts that actions (such as the execution of code) occur instantaneously; however, physical platforms obviously do not possess this property. This paper considers a scheduling problem that arises when one seeks to implement programs that are written under the synchrony assumption upon actual multiprocessor platforms, and proposes algorithms for solving this problem exactly and approximately.

Synchronous programming, Multiprocessor scheduling, Deadlines, Integer Linear Program, Approximation algorithm.

1 Introduction

The *Synchronous Reactive* model of computation (discussed in detail below) possesses many attractive features that makes it particularly suitable for specifying safety-critical cyber-physical systems. In recent work [4] we had investigated the issue of implementing systems that are so specified upon multiprocessor platforms, and had derived a polynomial-time approximation algorithm for obtaining implementations of synchronous reactive systems upon multiprocessors. In this paper we extend the work initiated in [4] by (i) further elaborating upon the relationship between implementation of synchronous reactive systems and multiprocessor scheduling; (ii) deriving (Section 4) an algorithm for solving the resulting multiprocessor scheduling problem exactly, and thereby assuring more resource-efficient implementations than was possible using the algorithm of [4]; and (iii) describing (Section 6) how this exact algorithm, which has exponential running time, may be used in conjunction with the polynomial-time algorithm.

Synchronous reactive (SR) systems. Most cyber-physical systems are *reactive* [10]: they repeatedly (i) monitor the environment within which they are operating via sensors; (ii) compute an appropriate response; and (iii) send signals to actuators to accomplish the computed response. For large and complex systems, it is often the case that abstractions such as periodic and sporadic jobs [15], [16] are at too "low" a level to enable a designer to specify such systems in a manner that inspires great confidence regarding the correctness of the specifications. Therefore the alternative of model-based design (MBD) techniques, which focus on provable correctness (at least, of functional properties), find widespread use in industry today. The abstractions underlying MBD techniques (such as the synchrony assumption [5], the actors abstraction [13], etc.) tend to place strong emphasis on formal methodologies and proof techniques. Large complex systems may be specified according to these abstractions and non-trivial functional correctness properties (such as safety, liveness, progress, etc.) of these systems proved.

Software modeling and development methodologies and commercial tools based on the *synchronous reactive* [10], [5], [9] model of computation are widely used in the model-based design and implementation of embedded systems. In this model of computation, larger systems are built by composing together individual *reactive components*, each of which maintains an internal state, and interacts with other components and the external environment via inputs and outputs in an ongoing manner: all components execute in a sequence of rounds with each reactive component reading its inputs, and computing outputs and updating its internal state based on the inputs and its current state, in each round. The semantics of SR systems are specified or formulated under an assumption called the synchrony hypothesis. As described by Benveniste and Berry [5, p. 1274], "the basic idea [underpinning the synchrony hypothesis] is very simple: we consider ideal reactive systems that produce their outputs synchronously with their inputs, their reaction taking no observable time." That is, SR systems are assumed to "produce their outputs synchronously with their inputs" [5, p. 1270]. The reaction intervals are thus reduced to reaction instants and do not overlap with each other. Hence the behavior of a system can be thought of as going through a potentially infinite series of steps, one occurring at each "logical" time-instant (often called a *tick* or a *round*): the system reads in its inputs at the time-instant corresponding to the t'th round and, based on its current state and these inputs, instantaneously computes the resulting outputs and updates its current state, and then does nothing until the time-instant corresponding to the $(t+1)$'th round. At this time-instant the system again reads in its inputs and instantaneously computes the resulting outputs and updates its state, and then waits until the time-instant corresponding to the $(t+2)$'th round, and so on.

The synchrony hypothesis facilitates the design process by permitting the system designer to focus on function-

ality and algorithmic issues, while abstracting away from implementation details. Additionally, it makes reasoning about concurrency a lot easier by eliminating the non-determinism resulting from interleaving of concurrent behaviors. This allows deterministic semantics, therefore making synchronous systems amenable to formal analysis and verification and thereby aiding in the process of obtaining statutory certification. These features help explain the immense popularity of software development methodologies based on the SR model of computation.

However, the undoubted benefits of synchronous models of computation do come at a price. The physical platforms on which systems are to be implemented do not, of course, satisfy the synchrony hypothesis: computations take (real) time to execute. Implementations of an SR model upon any particular computational platform must choose a time-unit large enough so that all the actions assumed to occur atomically at one instant complete execution upon the underlying platform strictly prior to the next instant[1]. Due to this and related factors, current implementation techniques for synchronous programs tend to make poor use of the platform resources. Although automated code-generators from synchronous programs have been developed,[2] the code produced by such automated code-generators tends to be rather inefficient. This is particularly true when compared to implementations of models that are specified using *job-based models* (see, e.g., [15], [16]), and scheduled using priority-based scheduling strategies such as RM or EDF [15]. This, then, is the trade-off involved in using SR-based design methodologies and tools in preference to earlier job-based ones: one gets an *easier to use and formally verify* methodology that, however, tends to make *less efficient use of resources*. As embedded systems have become increasingly more complex and difficult to design, this is a tradeoff that system designers have generally been willing to make; even more so since computational capabilities of computing platforms have, in keeping with the predictions of Moore's law, continued to increase at an exponential rate thereby making the efficiency issue less important. However, *energy* considerations and *thermal* issues are bringing efficiency considerations back to the forefront: even if plenty of computing capacity can be made available on a platform, providing the energy needed to enable all this computing capacity is fast becoming a bottleneck. This problem is further exacerbated in mobile platforms that are not tethered to the power-grid. The related problem of heat-dissipation in order to prevent inadmissible increases in the temperature of the platform is also often a major concern.

This Research. The work discussed in this paper should be viewed from this perspective of achieving resource-efficient implementations of reactive systems specified as synchronous reactive programs, upon implementation platforms comprising multiple processors that (obviously) do not satisfy the synchrony assumption. We report upon our investigations regarding one particular scheduling problem that appears central to efforts at doing so: *How should we schedule a collection of pieces of sequential code some or all of which are required to complete by specified deadlines and that is partially ordered with regards to the order in which the individual pieces of code may execute, upon an identical multiprocessor platform such that the number of processors needed is minimized?* We propose methods, based on prior techniques from scheduling theory, for solving this problem either exactly, via an algorithm that has running time exponential in the number of pieces of code in the collection, or approximately via a polynomial-time algorithm.

1. This requirement has been stated [7, p. 101] as the *bounded delay property* of the implementation: there is a maximum delay in completing the execution of the actions representing the system reaction to any input, which is strictly less than the minimum time that elapses between successive rounds.

2. E.g., Coder — https://www.mathworks.com/products/simulink-coder.html — generates standalone C and C++ code from Simulink models.

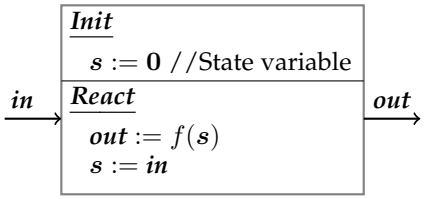


Fig. 1. An example Synchronous Reactive (SR) component (discussed, and used as an illustrative example, in Section 2).

Organization. The remainder of this paper is organized in the following manner. In Section 2 we briefly describe the synchronous reactive (SR) model of computation. In Section 3 we formally state the multiprocessor scheduling problem we are seeking to solve, and explain how this problem relates to the efficient multiprocessor implementation of SR programs. We derive algorithms in Sections 4 and 5 for solving this problem exactly and approximately, and characterize the effectiveness of the approximate algorithm (which has a smaller running time) vis-à-vis the exact one. In Section 6 we discuss how these two algorithms can be used together, with the approximate algorithm rapidly providing an initial starting point for the exact algorithm to then refine into an optimal solution. We conclude in Section 7 with a brief discussion placing the results presented here within the larger context of the design and implementation of SR programs, and a brief enumeration of some problems for future research.

2 The SR component Model

In this section we provide a brief introduction to the Synchronous Reactive (SR) component model upon which the remainder of this paper is based. Many different formalisms have been proposed for specifying SR computations, each with their own benefits and drawbacks; rather than being tied to any specific formalism, in this paper we will use the elegant abstract representation that is presented by Alur in the textbook [1]. In this model, an SR component

$$C = (I, O, S, Init, React)$$

is specified by

- the input variables I;
- the output variables O;
- the state variables S;

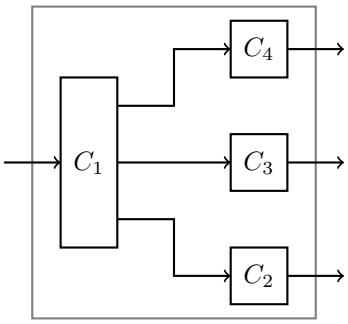


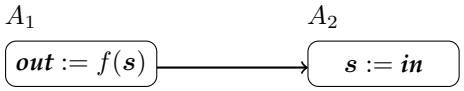
Fig. 2. Illustrating the processor of *composition* of synchronous reactive components – see the paragraph "**Composing components**" in Section 2. (Internal details of the three components are not depicted in this figure.)

- an initialization description *Init*, which specifies initial values for all the state variables; and
- a reaction description *React*, which defines the computation performed by the SR component during each round.

A simple example SR component is depicted in Figure 1. For this component, each of I, O, and S consists of a single element: $I = \{\boldsymbol{in}\}$, $O = \{\boldsymbol{out}\}$, and $S = \{\boldsymbol{s}\}$. (Note that each of ***in***, ***out***, and $\boldsymbol{s}$ may be equi-sized vectors.) The initialization description *Init* initializes the value of the state variable to equal $\mathbf{0}$, and the reaction description *React* asserts that in each round, the component (i) sets the value of its sole output variable ***out*** to be the result of applying some function $f(\cdot)$ to the current value of its sole state variable $\boldsymbol{s}$, and (ii) sets the value of $\boldsymbol{s}$ to be equal to the input that is read in at its sole input variable ***in*** during that round (and will therefore be used in computing the value of ***out*** in the next round). Hence if the values that are input to this component during rounds $0, 1, 2, \ldots$ are $\boldsymbol{x}_0, \boldsymbol{x}_1, \boldsymbol{x}_2, \ldots$, the values output by the component during rounds $0, 1, 2, \ldots$ are $f(\mathbf{0}), f(\boldsymbol{x}_0), f(\boldsymbol{x}_1), \ldots$; i.e., $f(\boldsymbol{x}_{i-1})$ is output during round i.

Composing components. Individual components of the kind illustrated in Figure 1 may be *composed* together by connecting the outputs of some components to inputs of other components. There are *component compatibility* rules that specify what compositions of components are permissible; i.e., do not lead to non-deterministic or ill-defined behaviors. It is not necessary for our purposes in this paper to know these rules in detail. Figure 2 illustrates an example composition, in which the three outputs of the SR component C_1 form the inputs to the three components C_2, C_3, and C_4 respectively. Under the synchronous reactive model of computation, the execution semantics of such compositions is that *all the components execute during each round*. A partial ordering of the execution of the components is implied by the connections between them[3] – e..g, in the example of Figure 2, component C_1 must execute before components C_2, C_3, and C_4 may begin to execute; components C_2, C_3, and C_4 may execute in parallel. Although the synchrony assumption models all this execution as occurring instantaneously (in "logical time"), realizations of this model must have the real-time duration of each round be no smaller than the actual execution duration of C_1 plus the largest of the actual execution durations of C_2, C_3, and C_4. Note that this is merely a lower bound on the duration of a round: the actual duration may need to be larger if adequate computational resources are not available to permit components C_2, C_3, and C_4 to execute in parallel.

Task-graph representation. The reaction description of an SR component may be considered to comprise several sequential *jobs*, that may have dependencies between them.[4] For instance, the reaction description of the SR component of Figure 1 may be split into two jobs A_1 and A_2, with A_1 responsible for execution of the first statement ("***out*** $:= f(\boldsymbol{s})$") and A_2 for the execution of the second statement ("$\boldsymbol{s}$:= ***in***"). Semantic considerations dictate that job A_1 complete execution before job A_2 begins to execute (since otherwise the value assigned to *out* during a round may incorrectly be computed using the current, rather than previous, round's input value): that is, there is a precedence constraint between these jobs:



Note that since A_1 is responsible for generating the output ***out*** of the component, this output becomes available the instant A_1 completes execution (even whilst A_2 is executing). Hence any other component that uses this output as an input may commence execution in parallel with the execution of A_2.

Task-graph representations of SR components responsible for performing non-trivial computations may be quite complex; Figure 3 depicts the task-graph representation of an SR component from [1, page 35], that has two inputs in_1 and in_2 and three outputs out_1, out_2, and out_3 . In this figure, the *read set* – variables that are read by the job – and the *write set* – variables that are written to by the job – are specified for each job A_i. (E.g., the notation states that the job A_3 reads in the variable x_1 and the input in_1, and writes to the variable x_1 and the output out_1.)

One of the component compatibility restrictions specified in [1] is that each output variable be in the write-set of exactly one job (this is indeed the case with the example

3. In many SR formalisms (including the one discussed in [1], the component compatibility rules would ensure that the input-to-output connections between components enforce such a partial ordering. Some formalisms, however, are based on *fix-point semantics* and do not require such a partial ordering to be enforced; they instead require that evaluation of all dependencies converge to a unique fix-point (see [6, p. 65] for an instructive description of the different approaches taken by different languages). We do not consider such fix-point based formalisms in this paper.

4. These jobs are called *tasks* in [1] – hence the term "task-graph representation". We prefer to refer to them as jobs here, since "tasks" typically denotes recurrent (periodic or sporadic) behavior in real-time scheduling theory.

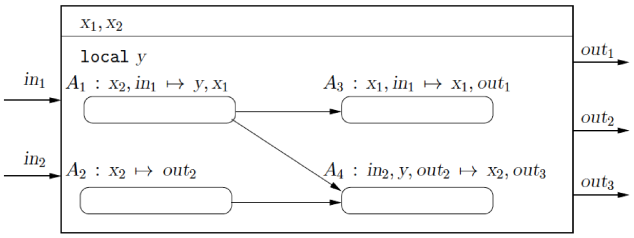


Fig. 3. An example task-graph representation of a single SR component.

component of Figure 3 — here, out_1 is in the write-set of job A_3, out_2 is in the write-set of job A_2, and out_3 is in the write-set of job A_4).

3 PROBLEM STATEMENT, AND PRIOR RESULTS

In this section we formally specify the scheduling problem that we will be exploring in greater detail in the remainder of this paper, and briefly review (in Section 3.1) some prior results from scheduling theory that we will be using.

Let us first motivate this problem by considering, once again, the composition of components that is depicted in Figure 2. With regards to this composition, let us suppose that

- We are given "black-box" implementations of the components C_2, C_3, and C_4, each executing upon its own dedicated computing platform; hence, the worst-case execution duration of each of these components is known (or can be determined using techniques of WCET-analysis [19]).
- The desired (real-time) duration of a round is known – this is presumably derived from the physics of the interaction of this system with the environment within which it is operating.[5]
- We have a task-graph representation of the component C_1 – i.e., a decomposition of its reaction description into sequential jobs, along with a partial ordering of these sequential jobs.

Our design objective is then to determine how much computational resources must be allocated to the component C_1, in order to guarantee to meet the specified round duration. In this paper, we restrict this decision to determining the number of processors that are to be assigned for the exclusive use of this component; in a more general setting, the computational resources under consideration may include communication bandwidth, storage memory, etc. See Section 7 for a discussion of some such generalizations.

Problem description. Generalizing from this example, the scheduling problem we seek to solve is this.

– We are ***given*** a directed acyclic graph $G = (V, E)$, which we seek to execute upon an identical multiprocessor platform under the preemptive global paradigm of multiprocessor scheduling. Each vertex $v \in V$ is characterized by a worst-case execution requirement $c(v)$. The vertices in some subset $O \subseteq V$ are each additionally characterized by a deadline $d(v)$; an overall deadline D is also specified.
– We seek to ***determine*** an implementation –a schedule– of the DAG upon the minimum number of processors that respects the precedence constraints and in which each vertex v with a deadline $d(v)$ completes within a duration $d(v)$ of the start of the schedule, and all vertices complete within a duration D of the start of the schedule.

That is, a problem ***instance*** is specified by

$$\Big\langle G = (V, E), c : V \to \mathbb{N}, d : V \to \mathbb{N}, D \Big\rangle, \tag{1}$$

(Here, we make the simplifying assumption that a deadline $d(v)$ is specified for all v: we can set $d(v) \leftarrow D$ for all v for which $d(v)$ is not explicitly specified.) *Given* such an instance, we seek to *determine* the smallest number m of processors upon which we can schedule the instance such that each vertex completes execution by its deadline. □

We now describe how the problem of efficiently implementing a given SR component can be reduced to an instance of this problem:

– The DAG $G = (V, E)$ is the task-graph representation of the reaction description of the SR component under consideration.
– The vertices V of this DAG represent the sequential jobs that comprise the reaction description; hence $c(v)$ denotes the WCET of the job modeled by vertex v.
– The graph edges E are precedence constraints enforcing the partial ordering on the jobs, as dictated by the reaction description.
– The overall deadline D denotes the duration of a round for the SR component.
– The vertices in O — those for which deadlines are specified — represent those jobs of the SR component that have the component's outputs in their write sets; if the output from a job is read in as input by some other component that has a WCET $\widetilde{C}$, then the deadline associated with this vertex is $(D - \widetilde{C})$.

Notice that if we were to execute each vertex upon a dedicated processor then we would require a total of $|V|$ processors; therefore the number of processors that is needed will never exceed $|V|$. (This is not to claim that any instance can be successfully scheduled on $|V|$ processors — it is possible, e.g., that some vertex v is the last vertex of a precedence-constrained sequence of vertices with cumulative execution requirement exceeding the deadline $d(v)$ of the vertex v.) Hence if we solve the problem of determining whether an instance can be successfully scheduled upon a given number m of processors, we could perform binary search over the range $[1, |V|]$ to determine the smallest value of m, by making $O(\log |V|)$ calls to this algorithm. In Sections 4 and 5 we will present exact (Section 4) and approximate (Section 5) algorithms for solving this problem.

3.1 Makespan minimization: Prior Results

We will see that determining whether an instance can be successfully scheduled upon a specified number of pro-

5. Suppose, for example, that the composite program is responsible for reading in some sensor input in an autonomous vehicle and determining whether to apply the emergency brakes: safety requirements derived from physical constraints such as vehicle speed and mass, inter-vehicular distance, etc., will specify the maximum duration of a reactive round of computation.

cessors is very closely related to the problem of scheduling precedence-constrained jobs to minimize makespan – commonly called the *makespan minimization problem*, which has been widely studied in the scheduling literature. We therefore start with some background on the makespan minimization problem. The makespan minimization problem seeks to obtain a schedule for any given precedence-constrained collection of jobs, each of which is characterized by an execution time (WCET) but no deadlines, upon an identical multicore platform in order to minimize makespan (for our purposes, the *makespan* of a schedule may be defined as the duration between the first and last instants at which execution occurs in the schedule). This problem has long been known [18] to be NP-hard in the strong sense, i.e., computationally highly intractable. However, Graham's *list scheduling* algorithm [8], which constructs a work-conserving schedule in a greedy manner by executing at each time instant an available job, if any are present, upon any available processor, performs reasonably well. It was shown [8] that list scheduling makes the following guarantee: if S_o denotes the smallest makespan with which a given precedence-constrained collection of jobs can be scheduled upon m processors by an optimal algorithm, then the schedule generated by list scheduling this collection of jobs upon m processors will have a makespan no larger than $(2 - \frac{1}{m}) \times S_o$. This result, in conjunction with a hardness result in [14] showing that determining a schedule with makespan $\leq \frac{4}{3}S_o$ remains NP-hard in the strong sense[6], suggests that list scheduling is a reasonable algorithm to use, and in fact most polynomial-time dynamic scheduling algorithms that are used for scheduling precedence-constrained collections of jobs upon multiprocessors use some variant or the other of list scheduling.

An upper bound on the makespan of a schedule generated by list scheduling is easily stated. Let W denote the cumulative worst-case execution time of all the jobs in a given precedence-constrained collection of jobs, and L denote the maximum cumulative worst-case execution time of any sequence of precedence-constrained jobs in the collection. It has been shown [8] that the makespan of the schedule generated by list scheduling upon m processors is guaranteed to be no larger than

$$\frac{W}{m} + L \times \left(1 - \frac{1}{m}\right)$$

4 An Exact Solution

In this section we present an exact algorithm for determining whether a problem instance of the kind specified as in Expression 1 in Section 3 above can be scheduled upon a given number of processors m. Our algorithm is centered on the idea of representing the problem instance as in Integer Linear Program (ILP). Determining whether an ILP has a feasible solution was one of the earliest problems shown to be NP-complete [12]. Indeed, it is known to be NP-complete *in the strong sense*; assuming P $\neq$ NP, this implies that ILP solvers with pseudo-polynomial running time cannot be developed. Despite this inherent intractability of ILP, however, the optimization community has devoted immense effort to devise extremely efficient implementations of ILP solvers, and highly-optimized libraries with such efficient implementations are widely available today in both open-source and commercial offerings. It is known that the duration taken by an ILP solver to solve a problem tends to correlate very strongly with the size of the problem to be solved — in particular, with the number of variables and constraints in the ILP that is being solved. Modern ILP solvers, particularly when running upon powerful computing clusters, are commonly capable of solving ILPs with tens of thousands of variables and constraints.

Overview of our approach. A wide variety of standard techniques have been developed within the traditional Operations Research (OR) community for ILP-based representations of scheduling problems (See, e.g, [3, Appendix C] for a text-book introduction to some of these techniques). Many of these techniques, including *time-indexing* (in which integer variables are used to represent which job is executing at each time-unit upon each processor) and *ordering* (in which integer variables are used to represent the order in which the jobs are to execute upon each processor), do not seem to be particularly suitable for our problem; e.g., time-indexing does not scale well since the number of integer variables needed is linear in the duration (makespan) of the schedule, while ordering is typically used for representing non-preemptive scheduling problems and it is not obvious how this technique should be extended to allow for preemption. Our ILP does not use either of these standard techniques, but is instead based on the following approach: given an instance as in Expression 1 with $n \stackrel{\text{def}}{=} |V|$

- Let f_i denote the completion-time of job v_i in some schedule, $1 \leq i \leq n$.
- The n completion-times $f_1, f_2, \ldots, f_n$ partition the makespan of the schedule into intervals. Let $\mathcal{I}_i$ denote the interval between f_i and the previous completion-time (with $\mathcal{I}_i = [0, f_i]$ if job v_i is the first one to complete execution).
- Let $c_{i,j}$ denote the amount of execution that job v_i receives in the interval $\mathcal{I}_j$.
- Our ILP will require, for each $i, 1 \leq i \leq n$ that
  1) $f_i \leq \min(D, d(v_i))$;
  2) $\sum_j c_{i,j} \geq c(v_i)$; and
  3) $c_{i,j} = 0$ for all j for which $\mathcal{I}_j$ is either after f_i or before the completion-times of all predecessor jobs of v_i.

Intuitively speaking, this approach requires that only polynomially many pieces of information be modeled in the ILP, thereby allowing it to be polynomial-sized. The precise manner in which this is achieved is detailed next, and is concurrently illustrated on a simple example instance depicted in Figure 4. The entire procedure is also summarized in pseudo-code form, in Figure 6.

We start out enumerating the **variables** in our ILP.

1) Positive real-valued completion-time variables $f_1, f_2, \ldots, f_{|V|}$, with f_i denoting the completion-time of the i'th job v_i.

6. In fact, assuming a reasonable complexity-theoretic conjecture that is somewhat stronger than P $\neq$ NP, a result of Svensson [17] implies that a polynomial-time algorithm for determining a schedule of makespan $\leq 2S_o$ for all m is ruled out.

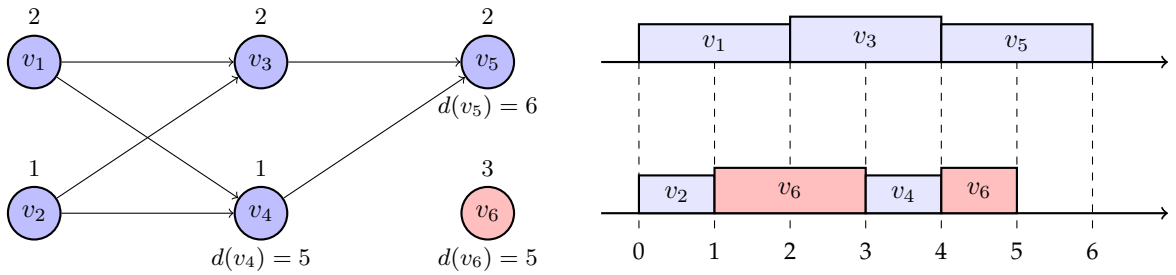


Fig. 4. An example instance, discussed in Section 4.. Numbers above the nodes denote the WCETs. A preemptive 2-processor schedule for the instance is depicted on the right. The execution of v_6 is depicted in a different color – note that v_6's execution is preempted at time 3 and its execution resumed at time 4.

2) Positive real-valued interval-duration variables I_j, for $1 \leq i \leq |V|$: the duration of the interval between f_j and the immediately-preceding completion-time. (I.e., $I_j = |\mathcal{I}_j|$, where $\mathcal{I}_j$ is as described in the overview of our algorithm above.)
3) Positive real-valued variables $c_{i,j}$ for $1 \leq i, j \leq |V|$, denoting the amount of execution v_i receives in interval $\mathcal{I}_j$.
4) "*Ordering*" variables $x_{i,j}$ for $1 \leq i, j \leq |V|$, that are restricted to take on a value $\in \{0, 1\}$, with the intended interpretation:

$$x_{i,j} = \begin{cases} 0, & \text{if } f_i \leq f_j \\ 1, & \text{if } f_i \geq f_j \end{cases} \tag{2}$$

We will see below how this intended interpretation is enforced by using a standard integer-programming technique.

Example 1. For the example instance of Figure 4, we would have six completion-time variables $f_1, f_2, \ldots, f_6$; six interval-duration variables $I_1, I_2, \ldots, I_6$; ($6 \times 6 =$) thirty-six $c_{i,j}$ variables; and thirty-six $x_{i,j}$ variables. For the schedule depicted in Figure 4, the completion-time variables would take on the values $f_1 = 2$, $f_2 = 1$, $f_3 = 4$, $f_4 = 4$, $f_5 = 6$, and $f_6 = 5$. The intervals, and their durations, would take on these values (also see Figure 5):

- $\mathcal{I}_1 = [0, f_2] = [0, 1]$, and hence $I_1 = 1$;
- $\mathcal{I}_2 = [f_2, f_1] = [1, 2]$, and hence $I_2 = 1$;
- $\mathcal{I}_3 = [f_1, f_3] = [2, 4]$, and hence $I_3 = 2$;
- $\mathcal{I}_4 = [f_3, f_4] = [4, 4]$, and hence $I_4 = 0$;
- $\mathcal{I}_5 = [f_4, f_6] = [4, 5]$, and hence $I_5 = 1$; and
- $\mathcal{I}_6 = [f_6, f_5] = [5, 6]$, and hence $I_6 = 1$.

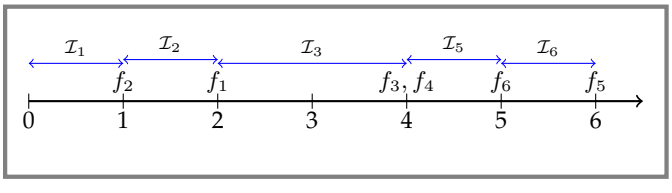


Fig. 5. Completion-times, and the intervals they define, for the example instance and schedule of Figure 4.

The $x_{i,j}$and $c_{i,j}$ variables would take on the following values:

$$x_{i,j} = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 \end{pmatrix} \quad c_{i,j} = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 \end{pmatrix}$$

We will not discuss the values assigned above to all of the thirty-six $c_{i,j}$ and thirty-six $x_{i,j}$ parameters ; instead, we explain a few example values below:

- $x_{1,2} = 1$ since $f_1 > f_2$.
- $x_{5,6} = 1$ since $f_5 > f_6$.
- $c_{6,1} = 0$ since v_6 executes for one time unit over the interval $\mathcal{I}_1 = [0, f_2] = [0, 1]$.
- $c_{6,2} = 1$ since v_6 executes for one time unit over the interval $\mathcal{I}_2 = [f_2, f_1] = [1, 2]$.
- $c_{6,3} = 1$ since v_6 executes for one time unit over the interval $\mathcal{I}_2 = [f_1, f_3] = [2, 4]$.
- $c_{6,4} = 0$ since v_6 executes for one time unit over the interval $\mathcal{I}_4 = [f_3, f_4] = [4, 4]$.
- $c_{6,5} = 1$ since v_6 executes for one time unit over the interval $\mathcal{I}_5 = [f_4, f_6] = [4, 5]$.
- $c_{6,6} = 0$ since v_6 executes for one time unit over the interval $\mathcal{I}_6 = [f_6, f_5] = [5, 6]$.

□

We next list the **constraints** in our ILP, for ensuring that the variables defined above do indeed take on their intended values:

1) As we had stated above, there is a standard technique for enforcing the intended interpretation on the ordering variables. Let M denote some large positive constant (for the purposes of our problem instances, it suffices to set M to have a value larger than D.) For each $i, j, 1 \leq i, j \leq |V|$, with $i \neq j$, we add the constraints

$$f_j \geq f_i - M \cdot x_{i,j} \tag{3}$$
$$f_i \geq f_j - M \cdot (1 - x_{i,j}) \tag{4}$$
$$1 = x_{i,j} + x_{j,i} \tag{5}$$

We can see why these constraints achieve the intended interpretation by considering the first two inequalities:

- If $x_{ij} = 0$, then the first inequality requires that $f_j \geq f_i - M \cdot 0$, i.e., $f_j \geq f_i$, as intended. The second

inequality requires that $f_i \geq f_j - M \cdot (1-0)$, i.e., $f_i \geq f_j - M$ (and since M is assumed to be a very large positive integer, this inequality does not constrain the possible value that may be assigned to f_i).

- If $x_{ij} = 1$, then the first inequality requires that $f_j \geq f_i - M$; since M is assumed to be a very large positive integer, this inequality does not constrain the possible value that may be assigned to f_j. However, the second inequality requires that $f_i \geq f_j - M \cdot (1-1)$, i.e., $f_i \geq f_j$, as intended.

The third constraint above simply ensures that exactly one of the two orderings is specified (and therefore ensures a total ordering on the completion-time values.)

2) We constrain each completion-time to be no larger than the corresponding deadline: for each i

$$f_i \leq d(v_i) \tag{6}$$

3) For each $j, 1 \leq j \leq |V|$, the interval-duration variable I_j must be constrained to be no larger than $(f_j - f_k)$ for each $f_k \leq f_j$; this is achieved by adding the n constraints

$$I_j \leq f_j - f_k + (M \cdot x_{kj}) \quad \text{for } k = 1, 2, \ldots, n \tag{7}$$

Note that the term within parenthesis (i.e., $(M \cdot x_{kj})$) evaluates to a very large positive integer if f_k is after f_j, and hence does not constrain the value of I_j.
In addition to the n constraints above, we must also consider the possibility that vertex v_j is the first one to complete execution; if so, there is no f_k constraining the value of f_j in the Constraints 7 above, and we instead have the Constraint

$$I_j \leq f_j \tag{8}$$

Example 2. For our example instance of Figure 4, the following seven constraints are placed upon the value of the interval duration I_4:

$$\begin{aligned}
I_4 &\leq f_4 - f_1 + M \cdot x_{1,4} \\
I_4 &\leq f_4 - f_2 + M \cdot x_{2,4} \\
I_4 &\leq f_4 - f_3 + M \cdot x_{3,4} \\
I_4 &\leq f_4 - f_4 + M \cdot x_{4,4} \\
I_4 &\leq f_4 - f_5 + M \cdot x_{5,4} \\
I_4 &\leq f_4 - f_6 + M \cdot x_{6,4} \\
I_4 &\leq f_4
\end{aligned}$$

(We point out that the fourth constraint above ($I_4 \leq f_4 - f_4 - M \cdot x_{4,4}$) is unnecessary and may be removed as an optimization step; however, retaining it has no effect on correctness.) □

4) The $c_{i,j}$ variables are constrained in several ways.

i) First for each i and j, $1 \leq i, j \leq |V|$, we have a constraint

$$c_{i,j} \leq M \cdot x_{i,j} \tag{9}$$

denoting that there is no benefit to executing job v_i after f_i. This is achieved by this constraint because $x_{i,j}$ equals 0 if $f_j > f_i$, and hence $c_{i,j}$ is constrained to be ≤ 0. (We note that this constraint has no effect when $f_k < f_i$ since $x_{i,j} = 1$ in that case and the constraint reduces to requiring that $c_{i,j}$ be no larger than the large positive number m.)

ii) Next for each i and j, $1 \leq i, j \leq |V|$, we have the constraint

$$c_{i,j} \leq I_j \tag{10}$$

denoting that vertex v_i must execute with no internal parallelism within interval $\mathcal{I}_j$.

iii) Next for each i, $1 \leq i \leq |V|$, we add a constraint that the total execution for job v_i must be enough to accommodate its wcet $c(v_i)$:

$$\sum_j c_{i,j} \geq c(v_i) \tag{11}$$

a) Finally, we must ensure that the cumulative execution over the j'th interval $\mathcal{I}_j$ does not exceed the cumulative capacity on the m processors during that interval: for each j, $1 \leq j \leq |V|$,

$$\sum_i c_{i,j} \leq m \cdot I_j \tag{12}$$

Example 3. We illustrate the role of Constraints (9) on our example instance of Figure 4. As shown in Figure 5, vertex v_6 completes before vertex v_5, and should therefore receive no execution during the interval $\mathcal{I}_6$ This is achieved by the constraint

$$\begin{aligned}
& c_{6,5} \leq M \cdot x_{6,5} \\
\Leftrightarrow \quad & c_{6,5} \leq M \cdot 0 \\
\Leftrightarrow \quad & c_{6,5} \leq 0
\end{aligned}$$

and hence the non-negative variable $c_{6,5}$ must take on value zero.
Also notice that Constraints (9) do not restrict vertex v_6 from executing in any of the earlier intervals. Consider, for instance, the interval $\mathcal{I}_3 = [2, 4]$:

$$\begin{aligned}
& c_{6,4} \leq M \cdot x_{6,4} \\
\Leftrightarrow \quad & c_{6,4} \leq M \cdot 1 \\
\Leftrightarrow \quad & c_{6,5} \leq M
\end{aligned}$$

which is essentially no constraint at all since M is a large positive number. □

5) Next we consider the edges. Each edge (v_j, v_i) constrains vertex v_i to only begin execution after vertex v_j has completed (i.e., after instant f_j). This is equivalent to stating that $c_{i,k}$ should equal zero for all k for which $f_k \leq f_j$; this is accomplished by the constraint

$$c_{i,j} \leq 0 \tag{13}$$

denoting that there is no benefit to assigning job v_j execution prior to f_j, the completion-time of its predecessor, and the following condition that extends this to intervals preceding the one that ends at f_j: for each k

$$c_{i,k} \leq M \cdot x_{k,j} \tag{14}$$

Example 4. Consider the edge (v_1, v_4) of the example instance of Figure 4. The constraint

$$c_{4,1} \leq 0$$

is explicitly added, thereby forbidding vertex v_4 from receiving execution in $\mathcal{I}_2 = [f_2, f_1]$. Additionally, Constraint (14) instantiated with $k = 2$:

$$\begin{aligned} & c_{4,2} \leq M \cdot x_{4,2} \\ \Leftrightarrow \quad & c_{4,2} \leq M \cdot 0 \\ \Leftrightarrow \quad & c_{4,2} \leq 0 \end{aligned}$$

ensures that vertex v_4 does not receive execution in $\mathcal{I}_1 = [0, f_2]$. We also note that Constraint (14) does not forbid vertex v_4 from executing *after* f_1; e.g., Constraint (14) instantiated with $k = 3$:

$$\begin{aligned} & c_{4,3} \leq M \cdot x_{4,3} \\ \Leftrightarrow \quad & c_{4,3} \leq M \cdot 1 \\ \Leftrightarrow \quad & c_{4,3} \leq M \end{aligned}$$

is a vacuous constraint. □

4.1 Running Time

As stated earlier in this section, solving ILPs is known [12] to be an NP-hard problem. However excellent tools, both commercial and open-source, have been developed that are able to solve fairly large ILPs in reasonable amounts of time. Hence it is important that ILPs be constructed be or "reasonable" size; the ILP we have constructed above is of size polynomial in the representation of the input instance:

- The number of *variables* is no more than $(2|V| + |V|^2)$ real-valued variables, and $|V|^2$ zero-one integer variables. Hence there is a total of $\Theta(|V|^2)$ variables.
- The number of *constraints* is no more than $4|V| + 6|V|^2 + |E| \times (1 + |V|) = \Theta(|V|(|V| + |E|))$.

5 A Polynomial-Time Algorithm

The ILP-based solution of Section 4 has worst-case running time exponential in the representation of the DAG. An algorithm was derived in [4] that solves the problem approximately. In this section we present a simplified version of the polynomial-time algorithm of [4]. This algorithm is presented in pseudo-code form in Figure 7; below, we will walk through the pseudo-code and thereby describe the algorithm.

The algorithm accepts as input the specifications $\langle G = (V, E), c : V \to \mathbb{N}, d : V \to \mathbb{N}, D \rangle$ of an instance and a number m of processors, and seeks to synthesize an m-processor schedule for the instance. It first (Line 1) topologically sorts [11] the vertices in the graph G, to obtain, in $\Theta(|V| + |E|)$ time, an ordering of the vertices such that for each edge (u, v) in the graph, vertex u appears before vertex v in the ordering. Next (Lines 2–4), it performs the "due-date modification" operation [2]: it modifies the deadline assigned to each vertex to be the smaller of its current value and the value that is implied by the deadlines of its successor vertices. Once all deadlines have been modified in this manner, the algorithm sorts the vertices into an ordered list L according to their (thus modified) deadline parameter values – Line 5.

Once this list L has been obtained, the algorithm simulates the preemptive list-scheduling of the jobs with the jobs ordered as listed in L (Line 6 of the pseudocode). If this schedule is correct (tested in Line 7) then it is returned as the desired implementation of the component (Line 8); else, failure is flagged (Line 9).

In order to determine whether the instance $\langle G = (V, E), c : V \to \mathbb{N}, d : V \to \mathbb{N}, D \rangle$ can be scheduled to meet all deadlines upon an m-processor platform, determine whether the following zero-one integer linear program is feasible:

Variables.

- For each $i, 1 \leq i \leq |V|$, non-negative real-valued variables f_i and I_i.
- For each i and j, $1 \leq i, j \leq |V|$, non-negative real-valued variable $c_{i,j}$ and zero-one integer variable $x_{i,j}$.

Constraints.

- For each $i, 1 \leq i \leq |V|$,

$$\begin{aligned} f_i &\leq d(v_i) \\ l_i &\leq f_i \\ \sum_j c_{i,j} &\geq c(v_i) \\ \sum_j c_{j,i} &\leq m \cdot I_i \end{aligned}$$

- For each i and j, $1 \leq i, j \leq |V|$,

$$\begin{aligned} f_j &\geq f_i - M \cdot x_{i,j} \\ f_i &\geq f_j - M \cdot (1 - x_{i,j}) \\ 1 &= x_{i,j} + x_{j,i} \\ I_i &\leq f_i - f_j + (M \cdot x_{ji}) \\ c_{i,j} &\leq M \cdot x_{i,j} \\ c_{i,j} &\leq I_j \end{aligned}$$

- For each edge (v_j, v_i):

$$c_{i,j} \leq 0$$

and for all k, $1 \leq k \leq |V|$,

$$c_{i,k} \leq M \cdot x_{k,j}$$

Fig. 6. ILP Representation of schedulability upon m processors

Difference with the algorithm in [4]. This algorithm differs from the one that was derived in [4] in the manner in which the ordered list L is obtained. As stated above, the algorithm of Figure 7 obtains the list by modifying the deadlines ("due dates") of all the vertices and sorting according to the modified deadlines. The algorithm in [4] does not perform the due-date modification: rather, it adopts a first principles approach of assigning priorities to job depending upon the deadlines (if any) of its successor jobs). While the end effect of the two algorithms is essentially equivalent, the approach of Figure 7 is simpler and more direct.

GENERATESCHEDULE$\Big(\Big\langle G = (V, E), c : V \to \mathbb{N}, d : V \to \mathbb{N}, D \Big\rangle, m\Big)$

1 Determine a topological ordering of the vertices
2 **for** each $v \in V$ considered in reverse topological order
3 **for** each edge $(v, u) \in E$
4 $d(v) \leftarrow \min(d(v), d(u) - c(u))$

5 $L \stackrel{\text{def}}{=}$ the jobs sorted in non-decreasing order of their $d(v)$ values // Takes $\Theta(|V| \log |V|)$ time
6 Simulate the preemptive list-scheduling of the jobs upon m processors, with the jobs ordered as in L
7 **if** each vertex v completes by its deadline $d(v)$
8 **return** the simulated schedule
else
9 **return** FAILURE

Fig. 7. Pseudo-code representation of the schedule-generation algorithm described in Section 5

5.1 Evaluation: Running Time

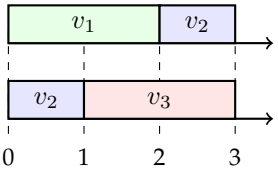
As briefly discussed above, the algorithm of Figure 7 can be implemented to have a running time that is a low-order polynomial in the number of vertices and edges of G, assuming a reasonably standard representation of its input. We briefly outline the running times of the individual parts of the pseudo-code:

1) Topological sorting (Line 1) takes $\Theta(|V| + |E|)$ time [11].
2) Due-date modification (Lines 2–4) requires that each vertex, and each edge, be looked at once – $\Theta(|V| + |E|)$ time.
3) Sorting the vertices (Line 5) can be done in $\Theta(|V| \log |V|)$ time.
4) Standard algorithms for simulating the preemptive list-scheduling of $|V|$ jobs on m processors have running-time linear in $|V| + m$.

5.2 Evaluation of Effectiveness

We saw above that in contrast to the ILP-based approach of Section 4, the algorithm of Figure 7 can be implemented to have a very efficient polynomial running time. However, it is not hard to show that the algorithm of Figure 7 is not optimal: this is illustrated in the following example.

Example 5. Consider an instance with three jobs v_1, v_2, and v_3, each with wcet equal to 2 (i.e., $c(v_1) = c(v_2) = c(v_3) = 2$), and deadline equal to 3 (i.e., $d(v_1) = d(v_2) = d(v_3) = 3$), that have no precedence constraints between themselves, and that is to be scheduled upon a 2-processor platform. Regardless of the value assigned to the list L in Line 5 of Figure 7, it may be verified that the vertex that is last in L will complete at time-instant 4 in the simulated list-scheduling of Line 6. However, an optimal preemptive schedule exists:



and, it may be verified, will be found by the ILP-based exact algorithm of Section 4. □

The example above serves as witness to the fact that the polynomial-time algorithm of Figure 7 is not optimal for determining schedulability. It turns out that we can quantify the "degree" to which it is not optimal. Recall that we'd mentioned earlier (in Section 3.1) the following result [8] for list scheduling. Let S_o denote the smallest makespan with which a given precedence-constrained collection of jobs can be scheduled upon m unit-speed processors by an optimal algorithm (e.g, the ILP-based one of Figure 6). The schedule generated by list scheduling this collection of jobs upon m unit-speed processors is guaranteed to have a makespan no larger than $(2 - \frac{1}{m}) \times S_o$. Equivalently, if the processors upon which the schedule generated by the list-scheduling algorithm is executed were each to have a speed $\geq (2 - 1/m)$, then the makespan of this schedule would not exceed the optimal makespan upon unit-speed processors. The following theorem, quantifying the relative performance of the approximation algorithm of Figure 7 vis-à-vis the optimal ILP-based algorithm, follows directly from this prior result:

Theorem 1. *If a component is correctly scheduled upon a platform comprising m unit-speed processors by the ILP-based algoritm of Section 4, then it is successfully scheduled by the algorithm of Figure 7 upon a platform comprising no more than m processors each of speed-$(2 - 1/m)$.* □

6 PUTTING THE PIECES TOGETHER

Recall that the algorithms of Section 4 and 5 above provide exact and approximate solutions to the problem of determining whether a component can be successfully scheduled upon a specified number of processors. However our objective is to determine the *smallest* number of processors $m_{\min}$ upon which the component can be implemented. This can be computed by

- doing binary search over the range $[1, |V|]$ to determine the smallest number of processors $\widehat{m}$ upon which the approximation algorithm is able to schedule the instance correctly; and then
- doing binary search over the range $[1, \widehat{m}]$ to determine the smallest number of processors $m_{\min}$ upon which the ILP is able to schedule the instance correctly.

In this manner the approximation algorithm rapidly provides a safe upper bound on the number of processors needed, and the ILP-based exact algorithm searches within the range demarcated by this safe upper bound to determine the actual optimal number of processors that are needed.

7 SUMMARY AND PERSPECTIVES

Research activities within the discipline of safety-critical real-time computing seem to be proceeding along (at least)

two distinct tracks. One, exemplified by the work in the Formal Methods and Model-Based Design (MBD) communities, seeks to define abstractions that enable the system designer to develop, and formally prove correctness of, large and complex systems. The other track, which includes the real-time scheduling, resource-allocation, the RTOS, etc. communities, is focused upon obtaining very efficient implementations of relatively simple abstractions, in a manner that is able to guarantee timing correctness (such as always meeting deadlines).

In this paper we report on some of our efforts at integrating these two tracks, by applying techniques of real-time scheduling theory to enhancing the efficiency of a popular family of MBD abstractions – those based on the Synchronous Reactive (SR) model of computation. We have illustrated how relatively well-known algorithms from multiprocessor real-time scheduling theory may be modified and adapted to render them applicable to SR component implementation.

The results reported here can be extended in several directions. We have assumed that we are focused upon the implementation of one particular component but the implementations of the remaining components is provided to us in immutable form. A natural generalization would be to develop an iterative process in which an implementation of each component is obtained (as described in this paper) by assuming that the others are provided, thereby seeking to minimize the total number of processors needed across all the components. Such a generalization also needs to consider situations in which components are connected in "chains" that are of length greater than two; for such situations, the problem of assigning intermediate deadlines is one that relates to some important fundamental problems in multiprocessor scheduling theory.

Acknowledgement. This work was supported in part by the National Science Foundation (NSF) under Grants CNS-1814739 and CPS-1932530,

REFERENCES

[1] Rajeev Alur. *Principles of cyber-physical systems*. MIT Press, 2015.

[2] Kenneth Baker and J. Bertrand. A dynamic priority rule for scheduling against due-dates. *Journal of Operations Management*, 3:37–42, 1982.

[3] Kenneth R. Baker and Dan Trietsch. *Principles of Sequencing and Scheduling*. Wiley Publishing, 2009.

[4] Sanjoy Baruah. The efficient multiprocessor implementation of synchronous reactive components. In *23rd IEEE International Symposium on Real-Time Distributed Computing, ISORC 2020*, 2020.

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

[6] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64 – 83, jan 2003.

[7] G. Berry. *The Esterel v5 Language Primer: version v5_91*. Centre de Mathématiques Appliquées, Ecole des Mines and INRIA, 2000.

[8] R. Graham. Bounds on multiprocessor timing anomalies. *SIAM Journal on Applied Mathematics*, 17:416–429, 1969.

[9] Nicolas Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Publisherss, 1993.

[10] David Harel and Amir Pnueli. On the development of reactive systems. In Krzysztof R. Apt, editor, *Logics and models of concurrent systems*, pages 477–498. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[11] A. B. Kahn. Topological sorting of large networks. *Communications of the ACM*, 5:558–562, 1962.

[12] R. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, 1972.

[13] Edward A. Lee, Stephen Neuendorffer, and Michael J. Wirthlin. Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems, and Computers*, 2, 2003.

[14] J. K. Lenstra and A. H. G. Rinnooy Kan. Complexity of scheduling under precedence constraints. *Operations Research*, 26(1):22–35, 1978.

[15] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[16] Aloysius Mok. *Fundamental Design Problems of Distributed Systems for The Hard-Real-Time Environment*. PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, 1983. Available as Technical Report No. MIT/LCS/TR-297.

[17] Ola Svensson. Conditional hardness of precedence constrained scheduling on identical machines. In *Proceedings of the 42nd ACM symposium on Theory of computing*, STOC '10, pages 745–754, New York, NY, USA, 2010. ACM.

[18] J. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384 – 393, 1975.

[19] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem – overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):36:1–36:53, May 2008.