Quasi-Static Scheduling of Communicating Tasks

Philippe Davoineau 1, Blaise Geneve 1, P.S. Thiagarajan 2, Shao Fa Yang 1

1 IRISA, CNRS & INRIA, Rennes, France
2 School of Computing, National University of Singapore

Abstract. Good scheduling policies for distributed embedded applications are required for meeting hard real time constraints and for optimizing the use of computational resources. We study the quasi-static scheduling problem in which (uncontrollable) control flow branchings can influence scheduling decisions at run-time. Our abstracted task model consists of a network of sequential processes that communicate via point-to-point buffers. In each round, the task gets activated by a request from the environment. When the task has finished computing the required responses, it reaches a pre-determined configuration and is ready to receive a new request from the environment. For such systems, we prove that determining existence of quasi-static scheduling policies is undecidable. However, we show that the problem is decidable for the important sub-class of “data branching” systems in which control flow branchings are due exclusively to data-dependent internal choices made by the sequential components. This decidability result—which is non-trivial to establish—exploits ideas derived from the Karp and Miller coverability tree [7] as well as the existential boundedness notion of languages of message sequence charts [5].

1 Introduction

We consider systems that consist of a finite collection of processes communicating via point-to-point buffers. Each process is a sequential transition system, in which non-deterministic branchings may be of two types: (i) a data-dependent internal choice made by a sequential component; (ii) a process waiting for messages on different input buffers. In the second case, the waiting process non-deterministically branches by picking up a message from one of the nonempty input buffers [3]. The system of processes is triggered by an environment iteratively in rounds. We model the system dynamics for just one round. It is easy to lift our results to multiple rounds. In each round, the environment sends a data item to one of the processes. This starts the computations done in the round. When the computation finishes, all the processes are in their final states and the buffers are empty. In a technical sense, buffers—viewed as counters without zero tests—are deployed here as over-approximated abstractions of FIFOs. We note that using FIFOs or zero tests would render the model Turing powerful [1].

In this setting, one is interested in determining a schedule for the processes. If at a configuration the scheduler picks the process $p$ to execute and $p$ is at a state with several outgoing transitions, then the schedule must allow all
possible choices to occur. As a result, such schedules are referred to as quasi-
static schedules. In addition, the schedule should never prevent the system from
(eventually) reaching the final state. We deem such schedules to be valid. In
addition, a quasi-static schedule is required to be regular in the sense that the
system under schedule should use only a bounded amount of memory to service
the request from the environment. In particular, the schedule should enforce a
uniform bound on the number of items stored in the buffers during the round.

Our first result is that determining whether a valid and regular quasi-static
schedule exists is undecidable. In fact the undecidability result holds even if the
system by itself is valid in that from every reachable global state it is possible to
reach the final global state the schedule does not need to enforce this. Next we
define data-branching systems in which the only branching allowed is local (da-
ta) branching; simultaneous polling on multiple input buffers is ruled out. We
show that for data-branching systems, one can effectively check whether there
exists a valid and regular quasi-static schedule. This result is obtained using
classical ideas from [7] and by exploiting a special scheduling policy, called the
canonical schedule. The canonical schedule is very similar to a normal form ob-
tained for determining the existential boundedness property of certain languages
of message sequence charts [5]. The crucial point here is that one cannot directly
apply the techniques of [7] because the canonical schedule uses zero tests on
buffers. Whereas, as is well known, it is often the case that zero tests lead to
undecidability.

Before considering related work, it is worth noting that our setting is strongly
oriented towards distributed tasks and their rounds-based executions. Hence it
does not cater for models capturing non-terminating computations such as Kahn
process networks [6]. At present, it is not clear whether our undecidability result
can be extended to such settings. Quasi-static scheduling (QSS) has been studied
before in a number of settings (see [8] for a survey). An early work in [2] studied
dynamic scheduling of boolean-controlled dataflow (BDF) graphs. Being Turing
powerful, the QSS problem for this class of systems is undecidable [2]. Later, [3]
proposed a heuristic to solve QSS on a different model called the YAPI model
by exploring only a subset of the infinite state space. There is however no proof
that the heuristic is complete even on a subset of YAPI models. The work [9]
considered QSS on a restricted class of Petri nets called Equal-Conflict Petri
nets and showed decidability. However the notion of schedulability used in [9] is
much weaker than the one in [3] or ours. Basically, under the scheduling regime
defined in [9], only a finite number of runs can arise, hence in effect, systems
with loops are not schedulable. In comparison, our system model is very close
to (general) Petri Nets. Our scheduling notion is essentially the one presented in
[3], slightly modified to fit our model. Our undecidability result is also harder to
obtain than the one in [2], since reachability is decidable for our model. Indeed,
the decidability of this quasi-static schedulability problem is stated as an open
problem in [3, 8]. The work [11] considered QSS with the setting of [3] and
proposed a sufficient (but not necessary) condition for non-schedulability based
on the structure of the Petri net system model.
In the next section we present our model and the quasi-static scheduling problem. Section 3 establishes the undecidability result in the general setting. Section 4 imposes the data-branching restriction and shows the decidability of the quasi-static scheduling problem under this restriction. The final section summarizes and discusses our results.

2 Preliminaries

Through the rest of the paper, we fix a finite set $\mathcal{P}$ of process names. Accordingly, we fix a finite set $\mathcal{C}$ of buffer names. To each buffer $c$, we associate a source process and a destination process, denoted $\text{src}(c)$ and $\text{dst}(c)$ respectively. We have $\text{src}(c) \neq \text{dst}(c)$ for each $c \in \mathcal{C}$. For each $p$, we set $\Sigma_p! = \{ c \mid c \in \mathcal{C}, \text{src}(c) = p \}$ and $\Sigma_p? = \{ c \mid c \in \mathcal{C}, \text{dst}(c) = p \}$. So, $c$ stands for the action that deposits one item into the buffer $c$ while $?c$ is the action that removes one item from $c$. For each $p$, we fix also a finite set $\Sigma_{p,cho}^c$ of choice actions. We assume that $\Sigma_{p,cho}^c \cap \Sigma_{q,cho}^c = \emptyset$ whenever $p \neq q$. Members of $\Sigma_{p,cho}^c$ will be used to label branches arising from abstraction of “if...then...else”, “switch...” and “while...” statements. For each $p$, we set $\Sigma_p = \Sigma_p! \cup \Sigma_p? \cup \Sigma_{p,cho}^c$. Note that $\Sigma_p \cap \Sigma_q = \emptyset$ whenever $p \neq q$. Finally, we fix $\Sigma = \bigcup_{p \in \mathcal{P}} \Sigma_p$.

A task system (abbreviated as “system” from now on) is a structure $A = \{(S_p, s_p^{init}, \rightarrow_p, s_p^f)\}_{p \in \mathcal{P}}$, where for each $p \in \mathcal{P}$, $S_p$ is a finite set of states, $s_p^{init}$ is the initial state, $\rightarrow_p \subseteq S_p \times \Sigma_p \times S_p$ is the transition relation, and $s_p^f$ is the final state. As usual, if $s_p \in S_p$ and $\delta = (s_p, a_p, s_p')$ is in $\rightarrow_p$ with $s_p' = s_p$, then we call $\delta$ an outgoing transition of $s_p$. We require the following conditions to be satisfied:

- For each $p \in \mathcal{P}$ and $s_p \in S_p$, if the set of outgoing transitions of $s_p$ is not empty, then exactly one of the following conditions holds:
  - Each outgoing transition of $s_p$ is in $S_p \times \Sigma_{p,cho}^c \times S_p$. Call such an $s_p$ a (data-dependent) choice state.
  - $s_p$ has precisely one outgoing transition, it is a send $(s_p, c, s_p')$, where $c \in \mathcal{C}, s_p' \in S_p$. Such an $s_p$ is called a sending state.
  - Each outgoing transition of $s_p$ is in $S_p \times \Sigma_p? \times S_p$. Call such an $s_p$ a polling state.

Intuitively, the system works in rounds. A round starts as if a message from the environment had just been received. At its final state, a process $p$ should stay watching buffers for messages possibly sent by other processes. If every process is in its final state, and all buffers are empty, a reset operation triggered by the environment may be performed to start a new round. This operation puts every process in its initial state from which the computation can start again. Thus, computations belonging to different rounds will not get mixed up. (We do not explicitly represent this reset operation in the system model.) For technical
convenience, we do not consider multi-rate communications, that is, multiple items can be deposited to or picked up from a buffer at one time. However, our results extend to multi-rate task systems easily.

For notational convenience, we shall assume that the system is deterministic, that is for each $p$, for each $s_p \in S_p$, if $(s_p, a_1, s_1 p)$, $(s_p, a_2, s_2 p)$ are in $\rightarrow_p$, then $a_1 = a_2$ implies $s_1 p = s_2 p$. All our results can be extended easily to non-deterministic systems. The dynamics of a system $A$ is defined by the transition system $TS_A$. A configuration is $(s, \chi)$ where $s \in \Pi_{p \in P} S_p$ and $\chi$ is a mapping assigning a non-negative integer to each buffer in $Ch$. We term members of $\Pi_{p \in P} S_p$ as global states. We view a member $s$ of $\Pi_{p \in P} S_p$ as a mapping from $P$ to $\bigcup_{p \in P} S_p$ such that $s(p) \in S_p$ for each $p$. When no confusion arises, we write $s_p$ for $s(p)$. The initial configuration is $(s^{init}, \chi^0)$ where $s^{init}(p) = s^{init}_p$ for each $p$.

Further, $\chi^0(c) = 0$ for every $c \in Ch$. We define $TS_A = (RC_A, (s^{init}, \chi^0), \Rightarrow_A)$ where the (possibly infinite) set $RC_A$ of reachable configurations and $\Rightarrow_A \subseteq RC_A \times \Sigma \times RC_A$ are the least sets satisfying the following:

- $(s^{init}, \chi^0) \in RC_A$.

- Suppose configuration $(s, \chi)$ is in $RC_A$ and $(s(p), a, s'_p) \in \rightarrow_p$ such that $a = ?_c$ implies $\chi(c) \geq 1$. Then configuration $(s', \chi') \in RC_A$ and $(s, a, (s', \chi')) \in \Rightarrow_A$, with $s'(p) = s'_p$, $s'(q) = s(q)$ for all $q \neq p$, and

  - If $a = !_c$, then $\chi'(c) = \chi(c) + 1$ and $\chi'(d) = \chi(d)$ for all $d \neq c$.
  - If $a = ?_c$, then $\chi'(c) = \chi(c) - 1$ and $\chi'(d) = \chi(d)$ for all $d \neq c$.
  - If $a \in \Sigma^{bho}$, then $\chi'(c) = \chi(c)$ for all $c \in Ch$.

We define $s^F$ to be the global state given by $s^F(p) = s^F_p$ for each $p$. We term $(s^F, \chi^0)$ as the final configuration.

We extend $\Rightarrow_A$ to $RC_A \times \Sigma^* \times RC_A$ in the obvious way and denote the extension also by $\Rightarrow_A$. Namely, firstly $(s, \chi) \Rightarrow_A (s, \chi)$ for any $(s, \chi)$ in $RC_A$. Secondly, if $(s, \chi) \Rightarrow_A (s', \chi')$ and $(s', \chi') \Rightarrow_A (s'', \chi'')$ where $s \in \Sigma^*$, $a \in \Sigma$, then $(s, \chi) \Rightarrow_{\Sigma^*} (s'', \chi'')$. A run of $A$ is a sequence $\sigma \in \Sigma^*$ such that $(s^{init}, \chi^0) \Rightarrow_{\Sigma^*} (s, \chi)$ for some $(s, \chi)$ in $RC_A$. We say that $\sigma$ ends at configuration $(s, \chi)$, and denote this configuration by $(\sigma^c, \chi^\sigma)$. We let $Run(A)$ denote the set of runs of $A$. The run $\sigma$ is complete iff $(\sigma^c, \chi^\sigma) = (s^F, \chi^0)$, and we denote by $Run_{\text{cpl}}(A)$ the set of complete runs of $A$.

Through the rest of this section, we fix a system $A$. We will often omit $A$ (e.g. write $RC, Run_{\text{cpl}}$ instead of $RC_A, Run_{\text{cpl}}(A)$). A configuration $(s, \chi)$ in

![Fig. 1. A task system with two processes $P_1, P_2.$](image)

4
RC is valid iff there exists \( \sigma \) with \( (s, \chi) \xrightarrow{\sigma} (s^\sigma, \chi^\sigma) \). A run \( \sigma \) is valid iff \( \sigma \) ends at a valid configuration. We say that \( \mathcal{A} \) is deadend-free iff every member of RC is valid. Note that one can effectively decide whether a given system is deadend-free by an easy reduction to the home marking reachability problem of Petri nets [4].

We show in Fig. 1 a system consisting of two processes \( P_1 \) and \( P_2 \) with \( c \) and \( e \) being buffers directed from \( P_1 \) to \( P_2 \) while \( o \) is a buffer directed from \( P_2 \) to \( P_1 \). The initial states are \( A \) and \( 1 \) while \( E \) and \( 3 \) are final states. The sequence \( b|e?e !o ?o \) is a complete run. The run \( \sigma = a|c|b|e?e !o ?o \) is not complete, even though \( s^\sigma = (E, 3) \). For, we have \( \chi^\sigma (c) = 1 \neq 0 \). This system is not deadend-free, since the run \( \sigma \) cannot be extended to a complete run.

2.1 Schedules

We now define the notion of schedule and schedulability. Let \( (s, \chi) \in RC \mathcal{A} \) be a reachable configuration. We say \( a \in \Sigma \) is enabled at \( (s, \chi) \) iff \( (s, \chi) \xrightarrow{a} (s', \chi') \) for some \( (s', \chi') \) in RC. We say that \( p \in \mathcal{P} \) is enabled at \( (s, \chi) \) iff some \( a \in \Sigma_p \) is enabled at \( (s, \chi) \). A schedule for \( \mathcal{A} \) is a partial function \( \text{Sch} \) from \( \text{Run} \) to \( \mathcal{P} \) which satisfies the following condition: \( \text{Sch}(\sigma) \) is defined if \( \sigma \) is enabled at \( (s^\sigma, \chi^\sigma) \), and if \( \text{Sch}(\sigma) = p \), then \( p \) is enabled at \( (s^\sigma, \chi^\sigma) \). Notice that if \( \sigma \) is complete, then no action is enabled at \( (s^\sigma, \chi^\sigma) \) and \( \text{Sch}(\sigma) = \epsilon \). For the schedule \( \text{Sch} \), we denote by \( \text{Run}/\text{Sch} \) the set of runs according to \( \text{Sch} \) and define it inductively as follows: \( \varepsilon \in \text{Run}/\text{Sch} \). If \( \sigma \in \text{Run}/\text{Sch} \), \( \text{Sch}(\sigma) = p, a \in \Sigma_p \) and \( \sigma a \) is a run, then \( \sigma a \in \text{Run}/\text{Sch} \). In particular, if \( \text{Sch}(\sigma) = p \) and \( \sigma \) can be extended by two actions \( a, b \) of process \( p \), then the schedule must allow both \( a \) and \( b \). It is easy to check that this definition of a schedule corresponds to the one in [3].

We say that the schedule \( \text{Sch} \) is valid for \( \mathcal{A} \) iff every run in \( \text{Run}/\text{Sch} \) can be extended in \( \text{Run}/\text{Sch} \cap \text{Run}_{\mathcal{A}} \). Next we define \( RC/\text{Sch} = \{(s^\sigma, \chi^\sigma) \mid \sigma \in \text{Run}/\text{Sch}\} \), the set of configurations reached via runs according to \( \text{Sch} \). We say that \( \text{Sch} \) is regular if \( RC/\text{Sch} \) is a finite set and \( \text{Run}/\text{Sch} \) is a regular language (in particular, the system under schedule can be described with finite memory). Finally, we say that \( \mathcal{A} \) is quasi-static schedulable (schedulable for short) iff there

---

**Fig. 2.** The system under schedule RC/Sch₂.
exists a valid and regular schedule for $A$. The quasi-static scheduling problem is to determine, given a system $A$, whether $A$ is schedulable. Again, it is easy to check that this definition of quasi-static schedulability corresponds to the one in [3]. In particular, the validity of the schedule corresponds to the fact that the system can always answer a query of the environment (by reaching the final configuration).

In the system of Fig. 1, the function $Sch_1(\sigma) = P$ with $P = P_1$ if $P_1$ is enabled at state $(s^e, \chi^e)$, $P = P_2$ otherwise, is a schedule. However, it is not regular, since $(a \mid c)^* \in \text{Run/Sch}_1$ goes through an unbounded number of configurations $((1, A), (n, 0, 0))$. On the other hand, the function $Sch_2(\sigma) = P$ with $P = P_2$ if $P_2$ is enabled at state $(s^e, \chi^e)$, $P = P_1$ otherwise is a valid and regular schedule. Fig. 2 shows the finite state space $RC/Sch_2$ which has no deadlock. In this figure, a configuration is of the form $XY \alpha \beta \gamma$, with $X \gamma$ the state of $P_2$ ($P_1$), and $\alpha, \beta, \gamma$ denote the contents of buffer $c, e$ and $o$ respectively. That is, the system of Fig. 1 is schedulable. Notice that a schedule does not need to prevent infinite runs. It just must allow every run to be completed.

3 General Case and Undecidability

The goal of this section is to establish the following result.

**Theorem 1.** The quasi-static scheduling problem is undecidable. In fact, it remains undecidable even when restricted to systems that are deadlock-free.

Our proof will consist of showing that the halting problem for deterministic two-counter machines can be uniformly reduced to our quasi-static scheduling problem. Given a deterministic two counter machine $M$, we shall construct a system $A$ such that $M$ halts if $A$ is schedulable.

To ease the presentation, we shall present the construction of $A$ in three phases and prove in each case that $M$ halts if $A$ is schedulable. In the first phase, our goal is to bring out the main ingredients of construction of $A$ with minimal amount of technical details. Thus, we shall allow transitions of $A$ to slightly deviate from the definition of system given in section 2. In the second phase, we modify the transitions of $A$ given in the first phase, so that they strictly adhere to the definition of system in section 2. In the first and second phase, the constructed $A$ needs not be deadlock-free. In the last phase, we show that the system $A$ constructed in the second phase can be in fact modified to be a system which is deadlock-free (and which strictly adheres to the definition of system in section 2).

More precisely, the constructed $A$ in the first two phases will have the following property: if $Sch$ is a valid schedule for $A$, then under $Sch$ the execution of $A$ will simulate the execution of $M$. Further, if the execution of $Sch$ leads $A$ to its final configuration, then in the corresponding execution $M$ will reach its halting state. We will show that whenever $M$ halts, $A$ has a valid schedule $Sch$. Further, $Sch$ must lead $A$ to its final configuration in a finite number of steps, hence it is a valid and regular schedule and $A$ turns out to be schedulable. On
the other hand, if $\mathcal{M}$ does not halt, it will turn out that $\mathcal{A}$ does not even have a valid schedule.

Let $C_1$, $C_2$ denote the two counters of $\mathcal{M}$. Let $\textit{halt}$ denote the halting state of $\mathcal{M}$. We assume that, for each control state $i$ other than $\textit{halt}$, the behaviour of $\mathcal{M}$ at $i$ is given by an instruction in one of the following forms with $j \in \{1, 2\}$:

- $(i, \text{Inc}(j), k)$: increment $C_j$ and move to control state $k$.
- $(i, \text{Dec}(j), k, m)$: if $C_j > 0$, then decrement $C_j$ and move to control state $k$; otherwise ($C_j = 0$), move to control state $m$.

Thus, $\mathcal{M}$ either stops at $\textit{halt}$ after a finite number of steps, or runs forever without visiting $\textit{halt}$.

Naturally, we encode counters of $\mathcal{M}$ by buffers of $\mathcal{A}$. Incrementing a counter of $\mathcal{M}$ amounts to sending a data item to the corresponding buffer. And decrementing a counter of $\mathcal{M}$ amounts to picking up a data item from the corresponding buffer. It is clear how the instruction $(i, \text{Inc}(j), k)$ of $\mathcal{M}$ can be simulated. The main difficulty is to simulate the instruction $(i, \text{Dec}(j), k, m)$. Indeed, in a system, a process can not branch to different states according to whether a buffer is empty or not. Further, when a schedule $\textit{Sch}$ selects a process $p$ to execute, $\textit{Sch}$ has to allow all transitions of $p$ that are enabled at the current state $s_p$ of $p$. However, the following observation will facilitate the simulation of an $(i, \text{Dec}(j), k, m)$ instruction. Suppose $s_p$ is a polling state with two outgoing transitions labelled $\textit{?a}$, $\textit{?b}$, where $\textit{src}(a) \neq \textit{src}(b)$. If prior to selecting $p$ and assuming both buffers $a$ and $b$ are currently empty, $\textit{Sch}$ can make the buffer $a$ nonempty (for example, by selecting $\textit{src}(a)$ to send a data item to $a$) and keep $b$ empty (for example, by not selecting $\textit{src}(b)$), then when $\textit{Sch}$ selects $p$, only the $?a$ transition is enabled and executed, while the $?b$ transition is ignored.

**Proof (of Theorem 1).** Let $\mathcal{M}$ be a deterministic two-counter machine as above with the associated notations. We construct a system $\mathcal{A}$ such that any valid schedule for $\mathcal{A}$ will guide $\mathcal{A}$ to simulate the execution of $\mathcal{M}$. As discussed above, one can then argue that $\mathcal{M}$ halts if $\mathcal{A}$ is schedulable. This will establish that the quasi-static scheduling problem is undecidable. To show that the undecidability remains even when restricted to systems that are deadend-free, we shall show that one can in fact modify $\mathcal{A}$ so that $\mathcal{A}$ is deadend-free. Further, any valid

![Diagram](image-url)

*Fig. 3. The architecture of $\mathcal{A}$*
and regular schedule for $\mathcal{A}$ will guide $\mathcal{A}$ simulate the execution of $\mathcal{M}$. One can then similarly argue that $\mathcal{M}$ halts if $\mathcal{A}$ is schedulable. This will then establish Theorem 1.

--- Phase (i): Here we construct a system $\mathcal{A}$ whose transitions slightly deviate from the definition of system in section 2. In particular, we shall allow a final state to be not a polling state and permit the outgoing transitions of a local state to consist of both receive transitions and choice transitions.

The system $\mathcal{A}$ has five processes $A, C(j), G(1), G(2), GZ$. Their communication architecture is illustrated in Fig. 3 where a label $ch$ on an arrow from process $p$ to process $q$ represents a buffer $ch$ with $srt(ch) = p$ and $dst(ch) = q$. For $j = 1, 2$, the number of items stored in buffer $c(j)$ will encode the value of counter $C_j$ of $\mathcal{M}$. Process $A$ will mimic the instructions of $\mathcal{M}$. For instructions of the form $(i, Inc(j), k)$, $A$ invoke $C(j)$ to increment $c(j)$. For instructions of the form $(i, Dec(j), k, m)$, $A$ allows to receive from both channel $gd$ (“Guess Dec”) and $gz$ (“Guess Zero”). The valid schedule will correctly simulates the emptiness test of buffer $c(j)$ by feeding the right channel $gd$ or $gz$. Figure 4 displays the transition systems of $GD, GZ,$ and $C(j), j = 1, 2,$ where an initial state is indicated by a pointing arrow, and a final state is drawn as a double circle. Figure 5 illustrates the transition system of $A$. For each $(i, Inc(j), k)$ instruction of $\mathcal{M}$, $A$ contains transitions shown in Fig. 5(i). For each $(i, Dec(j), k, m)$ instruction
of $M$, $A$ contains transitions shown in Fig. 5(ii), where the state sink is a distinguished state with no outgoing transitions. Unlabelled transitions represent those with labels in $\Sigma^{x\cdot\emptyset}$. For the halting state of $M$, $A$ contains special transitions shown in Fig. 5(iii), whose purpose is to empty the buffers $c(1), c(2)$ after $A$ reaches $halt$. The initial state of $A$ is the initial state of $M$, and the final state of $A$ is $halt$.

Let $Sch$ be a valid schedule for $A$. Suppose that, according to $Sch$, execution of $A$ arrives at a configuration in which $A$ is at state $i$. There are two cases to consider:

—Case (i): The corresponding instruction of $M$ is $(i, Inc(j), k)$.

It is easy to see that $Sch$ has no choice but to select $A$ to execute $!inc(j)$, then select $C(j)$ three times to execute $?inc(j), !c(j), !inc-ok(j)$, and finally select $A$ to execute $?inc-ok(j)$. In doing so, $c(j)$ is incremented and $A$ moves to state $k$.

—Case (ii): The corresponding instruction of $M$ is $(i, Dec(j), k, m)$.

Note that from state $i$ of $A$, there are two outgoing transitions labelled $?gd$, $?gz$ respectively. Consider first the case where $c(j)$ is greater than zero. We argue that $Sch$ has to guide $A$ to execute only the transition $?gd$ in order to be valid. That is, $Sch$ should ensure that the $?gd$ transition of $A$ is enabled by selecting GD. It must further ensure that the $?gz$ transition of $A$ is not enabled which can be done by not scheduling the process GZ. By doing so, $c(j)$ will be decremented and $A$ will move to state $k$. If on the other hand, $?gz$ is enabled while $c(j)$ is greater than zero, then $Sch$ will allow $A$ to take the $?gz$ transition. Consequently, $Sch$ will allow $A$ to reach state $m$, as well as state sink. As sink has no outgoing transitions, the run which leads $A$ to sink is not valid. This however will contradict the hypothesis that $Sch$ is valid.

Similarly, for the case where $c(j)$ is zero, it is easy to see that $Sch$ has to guide $A$ to execute only $?gz$. Further, after executing the $?gz$ transition, $A$ will move to state $m$ only, since the corresponding $?c(j)$ transition will not be enabled.

We claim that $M$ halts iff $A$ is schedulable. To see this, suppose $M$ halts. Then from the above argument that $M$ may be simulated by executing $A$ under a valid schedule, it is easy to construct a valid schedule $Sch$ for $A$ so that $Sch$ will lead $A$ to the configuration in which each process is at its final state, and all buffers except possibly $c(1), c(2)$ are empty. From Fig. 5(iii), it follows that $Sch$ will eventually also empty $c(1), c(2)$. Further, it also follows that $Sch$ is regular and thus $A$ is schedulable.

Suppose $M$ does not halt. Assume further that $Sch$ is a valid schedule for $A$. Then as explained above, $Sch$ simulates the execution of $M$ and thus process $A$ can never reach its final state $halt$. Thus $Sch$ can not be valid, a contradiction.

—End of Phase (i)

—Phase (ii): In this phase, we modify the transitions of $A$ in phase (i) so that they strictly adhere to the definition of system in section 2.

Firstly, we modify the communication architecture of processes of $A$ to be as displayed in Fig. 6. The transition systems of GD, GZ and processes $C(j)$, $j = 1, 2$, are shown in Fig. 7. Note that the final states of processes GD, GZ are now polling states. For $j = 1, 2$, process $C(j)$ is constructed in the same
way as in phase (i). The transition system of $A$ is illustrated in Fig. 8. For each 
(i, $Inc(j), k$) instruction of $M$, $A$ contains transitions shown in Fig. 8(i). For each 
(i, $Dec(j), k, m$) instruction of $M$, $A$ contains transitions shown in Fig. 8(ii), 
where the state $sink$ is a distinguished state with no outgoing transitions. As 
in phase (i), for the halting state of $M$, $A$ contains special transitions shown 
in Fig. 8(iii). It is clear that the transitions of $A$ now strictly adhere to the 
definition of system in section 2.

Let $Sch$ be a valid schedule for $A$. As in Case (i), we argue that $Sch$ will guide 
$A$ to simulate the execution of $M$. The simulation of an (i, $Inc(j), k$) instruction 
is as in Case (i). Now suppose that, according to $Sch$, execution of $A$ arrives 
at a configuration in which $A$ is at state $i$ and the corresponding instruction 
of $M$ is (i, $Dec(j), k, m$). And each of $GD, GZ$ is at its initial state. Then it is 
not difficult to see that $Sch$ must first select $A$ twice to execute !$make - gd$, 
!$make - gz$ transitions and thus $GD, GZ$ become enabled. Next, suppose $c(j)$ 
is greater than zero. Then as in phase (i), $Sch$ has to guide $A$ to execute only 
the transition ?$gd$. And eventually, $c(j)$ is decremented, $A$ moves to state $k$, and 
$GD, GZ$ return to their initial states. On the other hand, if $c(j)$ is zero, then $Sch$ 
has to guide $A$ to execute only the transition ?$gz$. And eventually, $c(j)$ remains 
zero, $A$ moves to state $m$, and $GD, GZ$ return to their initial states.
With the observation that any valid schedule \( \text{Sch} \) will guide \( \mathcal{A} \) to simulate the execution of \( \mathcal{M} \), it follows from similar arguments as in case (i) that \( \mathcal{M} \) halts iff \( \mathcal{A} \) is schedulable.

---End of Phase (ii)

---Phase (iii): Finally, we modify the construction of \( \mathcal{A} \) in phase (ii) so that \( \mathcal{A} \) is in fact deadend-free. Further, we will construct \( \mathcal{A} \) in such a way that any valid and regular schedule for \( \mathcal{A} \) will simulate the execution of \( \mathcal{M} \). One can then show that \( \mathcal{M} \) halts iff \( \mathcal{A} \) is schedulable. In what follows, we first explain the construction of \( \mathcal{A} \), then argue that \( \mathcal{M} \) halts iff \( \mathcal{A} \) is schedulable, and finally show that \( \mathcal{A} \) is in fact deadend-free.

The communication architecture of \( \mathcal{A} \) is now as shown in Fig. 9. The transition systems of \( \mathcal{GD}, \mathcal{GZ} \) and processes \( C(j), j = 1, 2 \), are displayed in Fig. 10. The transition system of \( \mathcal{A} \) is illustrated in Fig. 11. For each \((i, \text{Inc}(j), k)\) instruction of \( \mathcal{M}, \mathcal{A} \) contains transitions shown in Fig. 11(i). For each \((i, \text{Dec}(j), k, m)\) instruction of \( \mathcal{M}, \mathcal{A} \) contains transitions shown in Fig. 11(ii), where \( \text{sink} \) is a distinguished state. For the states \( \text{sink} \) and \( \text{halt} \), \( \mathcal{A} \) contains special transitions shown in Fig. 11(iii) (where unlabelled arrows represent transitions with labels in \( \Sigma^{(k \circ)} \)).

We first note that the special transitions in Fig. 11(iii) are designed in such a way that any valid and regular schedule must never lead \( \mathcal{A} \) to a configuration in which \( \mathcal{A} \) is at the state \( \text{sink} \). To see this, suppose \( \text{Sch} \) is a valid and regular sched-
ule for A. Assume further that according to Sch, A arrives at a configuration in which A is at sink. Note that Sch can not discriminate between the two outgoing transitions of sink which are data-dependent choice transitions. Thus, it is not difficult to see that Sch has to admit runs in which the transitions !inc(1), ?inc-ok(1) of Fig. 11(iii) can be executed arbitrarily many times (with transitions ?inc(1), !c(1), !inc-ok(1) from C(1) being interleaved). That is, Sch will admit complete runs which pass configurations with A being at state sink and the size of c(1) can be arbitrarily large. Consequently, Sch is not regular, a contradiction.

By the above observation that any valid and regular schedule for A must guide A to avoid visiting sink, one can use similar arguments as in phase (ii) to show that any valid and regular schedule for A will guide the execution of A to simulate the execution of M. Now, similar to phase (i), if M halts, then one can easily construct a valid and regular schedule which leads A to the configuration in which each process is at its final state, and all buffers except possibly c(1), c(2) are empty. Further, during the execution of A under Sch, A never visits state sink. With the special transitions shown in Fig. 11(iii), Sch will eventually also empty buffers c(1), c(2). Thus A is schedulable.

On the other hand, if M does not halt, as any valid schedule needs to reach the halt state in process A, any valid schedule needs to go through the sink

\[ C(1) \rightarrow \text{inc}(1) \leftarrow A \rightarrow \text{inc}(2) \rightarrow C(2) \]

\[ \text{gd, gd-wrong} \rightarrow \text{make-gd} \]

\[ \text{ gz, gz-ok} \rightarrow \text{make-gz} \]

**Fig. 9. The architecture of A in phase (iii)**

\[ ?\text{inc}(j) \rightarrow \text{inc}-\text{ok}(j) \]

\[ \text{lc}(j) \rightarrow \text{make-gd} \]

\[ \text{lgd-wrong} \rightarrow \text{lgd} \]

\[ \text{?make-gz} \rightarrow \text{lgz-ok} \]

**Fig. 10. Description of processes GD, GZ, C(j) in phase (iii)**
state, and hence it is not regular. That is, \( \mathcal{A} \) does not have a valid and regular schedule. We have now shown that \( M \) halts iff \( \mathcal{A} \) is schedulable.

Finally, we argue that the system \( \mathcal{A} \) constructed in this phase is in fact deadend-free. We shall assume that from any control state \( i \) of \( M \) except the halting state, it is possible to reach a control state \( t \) whose corresponding instruction has the form \( (i, \text{Dec}(j), k, m) \). We note that this assumption involves no loss of generality, since one may replace each \((i, \text{Inc}(j), k)\) instruction equivalently by the collection of three instructions \((i, \text{Inc}(j)), (i', \text{Inc}(j)), (i'', \text{Dec}(j), k, k)\) where \( i', i'' \) are newly created control states with \( i' \neq i'' \).

To show that \( \mathcal{A} \) is deadend-free, we need to argue that every run \( \sigma \) of \( \mathcal{A} \) can be extended to a complete run. Loosely speaking, it suffices to consider two types of runs of \( \mathcal{A} \):

**Type I:** Runs which simulate the execution of \( M \) and never visit a configuration with \( A \) being at state sink.

**Type II:** Runs which end at the configuration with \( A \) being at state sink, every other process being at its initial state, and all buffers except possibly \( c(1), c(2) \) being empty.
Now we show that $A$ is deadend-free by considering two cases according to whether $M$ halts or not.

—Case (i): $M$ halts.

Let $\sigma$ be a run of $A$. If $\sigma$ is of type I, then clearly $\sigma$ can be extended to a complete run of $A$. If $\sigma$ is of type II, then it is easy to see that $\sigma$ can be extended to a run ending at the configuration in which $A$ is at state $halt$, every other process is at its initial state, and all buffers except possibly $c(1), c(2)$ are empty. It follows that $\sigma$ can be extended to a complete run.

—Case (ii): $M$ does not halt.

Let $\sigma$ be a run of $A$. First consider the case where $\sigma$ is of type I. As discussed above, we can assume from any control state $i$ of $M$ except the halting state, it is possible to reach a control state $i$ whose corresponding instruction has the form $(i, Dec(j), k, m)$. Thus, $\sigma$ can be extended to a run $\sigma'$ where $\sigma'$ ends at a configuration in which $A$ is at some state $i$ whose corresponding instruction of $M$ is of the form $(i, Dec(j), k, m)$. From the transitions shown in Fig. 11(ii), it is easy to see that $\sigma'$ can be further extended to a run $\sigma''$ where $\sigma''$ ends at a configuration in which $A$ is at state sink. Further, $\sigma''$ can be extended to a complete run.

For the case where $\sigma$ is of type II, by similar arguments as in case (i), one sees that $\sigma$ can be extended to a complete run.

—End of Phase (iii)

With the construction of a deadend-free system $A$ in phase (iii) and the corresponding arguments that $M$ halts iff $A$ is schedulable, we complete the proof of Theorem 1.

4 Data-Branching and Decidability.

We have observed that a schedule’s ability to indirectly discriminate between two receive actions (e.g. ?gd and ?gz) of the same process is crucial to our undecidability proof. The question arises whether the quasi-static scheduling problem for systems in which such choices are not available is decidable. We show here that the answer is indeed yes. In this context, we wish to emphasize that the definition of quasi static scheduling used in [9] will permit only a finite collection of runs and hence does not cater for systems with internal loops. Thus, the problem solved in [9] is simpler than the one addressed here.

The system $A$ is said to be data-branching iff for each $p$, for each $s_p \in S_p$, if $s_p$ is a polling state, then it has exactly one outgoing transition. Thus the only branching states are those at which internal data-dependent choices take place.

**Theorem 2.** Given a data-branching system $A$, one can effectively determine whether $A$ is schedulable.

The rest of this section is devoted to the proof of theorem 2. We shall assume throughout that $A$ is data-branching. The proof relies crucially on the notion of
a canonical schedule for $A$, denoted $\text{Sch}_{ca}$. The canonical schedule is positional, that is, $\text{Sch}_{ca}(\sigma) = \text{Sch}_{ca}(\sigma')$ whenever runs $\sigma, \sigma'$ end at the same configuration. Thus, we shall view $\text{Sch}_{ca}$ as a function from $\text{RC}$ to $\mathcal{P}$. Informally, at configuration $(s, \chi)$, if there is a $p \in \mathcal{P}$ such that $p$ is enabled and $\chi$ has exactly one outgoing transition of $p$, then $\text{Sch}_{ca}$ is a valid and regular schedule for $A$. We prove that one can effectively decide whether $\text{Sch}_{ca}$ is a valid and regular schedule (Thm. 9).

4.1 The Canonical Schedule.

We fix a total order $\leq p$ on $\mathcal{P}$ and define the canonical schedule $\text{Sch}_{ca}$ for $A$ as follows. For each configuration $(s, \chi)$, let $P(\sigma, \chi)_\text{enable}$ be the set of processes enabled at $(s, \chi)$. We partition $P(\sigma, \chi)_\text{enable}$ into $P(\sigma, \chi)_\text{poll}$, $P(\sigma, \chi)_\text{choice}$, and $P(\sigma, \chi)_\text{send}$ as follows. For $p \in P(\sigma, \chi)_\text{poll}$, we have: (i) $p \in P(\sigma, \chi)_\text{poll}$ if $s_p$ is a polling state; (ii) $p \in P(\sigma, \chi)_\text{choice}$ if $s_p$ is a choice state; (iii) $p \in P(\sigma, \chi)_\text{send}$ if $s_p$ is a sending state. We further define the set $P(\sigma, \chi)_\text{send-min} \subseteq P(\sigma, \chi)_\text{send}$ as follows: for $p \in P(\sigma, \chi)_\text{send}$, we have $p \in P(\sigma, \chi)_\text{send-min}$ if $\chi(c_p) \leq \chi(c_q)$ for each $q \in P(\sigma, \chi)_\text{send}$, where $c_p$ (respectively, $c_q$) is the action of $p$ (respectively, of $q$) enabled at $(s, \chi)$.

The canonical schedule $\text{Sch}_{ca}$ maps each configuration $(s, \chi)$ to the process $\text{Sch}_{ca}(s, \chi)$ as follows. If $P(\sigma, \chi)_\text{poll} \cup P(\sigma, \chi)_\text{choice} \neq \emptyset$, then $\text{Sch}_{ca}(s, \chi)$ is the least member of $P(\sigma, \chi)_\text{poll} \cup P(\sigma, \chi)_\text{choice}$ with respect to $\leq p$. Otherwise, $\text{Sch}_{ca}(s, \chi)$ is the least member of $P(\sigma, \chi)_\text{send-min}$ with respect to $\leq p$. It is straightforward to verify that $\text{Sch}_{ca}$ adheres to the definition of schedule.

**Proposition 3.** A data-branching system $A$ is schedulable iff $\text{Sch}_{ca}$ is a valid and regular schedule for $A$.

To facilitate the proof of Prop. 3, we introduce now an equivalence on complete runs. For $\sigma \in \Sigma^*$ and $p \in \mathcal{P}$, let $\text{pr}_p(\sigma)$ be the sequence obtained from $\sigma$ by erasing letters not in $\Sigma_p$. We define the equivalence relation $\sim \subseteq \text{Run}_{cpl} \times \text{Run}_{cpl}$ as follows: $\sigma \sim \sigma'$ iff for every $p \in \mathcal{P}$, $\text{pr}_p(\sigma) = \text{pr}_p(\sigma')$. We note a useful relation between $\sim$ and schedules.

**Observation 4.** Let $\sigma$ be a complete run of a data-branching system $A$. Suppose that $\text{Sch}$ is a schedule of $A$ (not necessarily valid nor regular). Then there exists a complete run $\sigma'$ such that $\sigma' \sim \sigma$ and $\sigma' \in \text{Run}/\text{Sch}$.

**Proof.** Let $\sigma = \tau a \tau'$, with $a \in \Sigma_p$, $\tau \in \text{Run}/\text{Sch}$, and $\text{Sch}(\tau) = q \neq p$. In particular, $\tau a \notin \text{Run}/\text{Sch}$ and $q$ is enabled at $(s', \chi')$. We show that there exists a complete $w$ of the form $\tau b c$ with $b \in \Sigma_q$ (thus $\tau b$ is according to $\text{Sch}$) and
$w \sim \sigma$. Repeating inductively this argument then eventually yields the desired complete run $\sigma'$ according to $\text{Sch}$ with $\sigma' \sim \sigma$.

Now we show the existence of $w$ above by considering two cases. Note that $s_q^\sigma$ is the final state of $q$. It thus follows from the definition of a task system that, either $s_q^\sigma$ has no outgoing transitions, or $s_q^\sigma$ is a polling state.

— **Case (i):** $s_q^\sigma$ is a sending state or a choice state.

We have $s_q^\tau \neq s_q^\sigma$ since $s_q^\sigma$ either has no outgoing transition or is a polling state. So some choice or sending action $b$ in $\Sigma_q$ should occur in $\tau'$ to move from $s_q^\tau$. Hence, let $\tau' = \rho b \rho'$ where $\rho$ contains no letter of $\Sigma_q$. Then one readily verifies that $w = \tau bapp' \sigma$ is also a run of $A$ and $w \sim \sigma$.

— **Case (ii):** $s_q^\tau$ is a polling state.

Since $\text{Sch}(\tau) = q$, some action $?c$ with $\text{dst}(c) = q$ is enabled at the configuration $(s^\tau, \chi^\tau)$. That is, $(s_q^\tau, ?c, s_q^\sigma)$ is an outgoing transition of $s_q^\sigma$ and $\chi^\tau(c) > 0$. We show that $?c$ occurs in $\tau'$ and thus we can write $\tau'$ in the form of $\rho ?c \rho'$ where $\rho$ contains no letter of $\Sigma_q$. It easily implies that $w = \tau ?capp' \sigma$ is also a run of $A$ and $w \sim \sigma$. First, since $A$ is data-branching and $s_q^\sigma$ is a polling state, if there is an action in $\Sigma_q$ in $\tau'$, then the first such action must be $?c$. Then, by contradiction, if there is no action on $q$ in $\tau'$, then $s_q^\sigma = s_q^\sigma'$, and in particular there is no $?c$ in $\tau'$ (since it is an action of $q$). Hence $\chi^\sigma(c) \geq \chi^\sigma(c) > 0$ contradicting the fact that $(s^\sigma, \chi^\sigma)$ is a final configuration. \hfill \Box

Observation 4 implies that a run $\sigma$ of $\text{Run}/\text{Sch}$ can be extended to a run in $\text{Run}_{\text{opt}}/\text{Sch}$ if it can be extended to a run in $\text{Run}_{\text{opt}}$. This holds for every schedule $\text{Sch}$ (not necessarily valid nor regular), provided the system is data-branching. Using this observation, we can now prove that if there exists a valid schedule, then $\text{Sch}_{\text{ca}}$ is valid too.

**Lemma 5.** A data-branching system $A$ admits some valid schedule iff $\text{Sch}_{\text{ca}}$ is valid for $A$.

**Proof.** It suffices to consider the “only if” direction. For contradiction, take $\sigma a \in \text{Run}/\text{Sch}_{\text{ca}}$ such that $a \in \Sigma_p$, and there exists a run $\tau$ with $\sigma \tau \in \text{Run}_{\text{opt}}$, but $\sigma \tau \neq ?c \in \text{Run}_{\text{opt}}$ for any $\tau$. Since $\sigma \tau \in \text{Run}_{\text{opt}}$ by observation 4, there exists $\sigma' \sim \sigma \tau$ and $\sigma' \in \text{Run}_{\text{opt}}/\text{Sch}$. Note that $a \in \Sigma_p$ is enabled at $(s^\sigma, \chi^\sigma)$.

Suppose that $s_p^\sigma$ is a choice or sending state. This implies that $s_p^\sigma \neq s_p^\sigma$ and hence there exists a choice or sending action $b \in \Sigma_p$ with $\tau = \tau_1 b \tau_2$, where $\tau_1$ has no action belonging to $p$. This implies that $\sigma' \sim \sigma \tau_1 b \tau_2$. Let us decompose $\sigma'$ as $\sigma' = \tau_1' b \tau_2'$, with as many actions belonging to $p$ in $\tau_1'$ as in $\tau_1$. We have $\text{Sch}(\tau_1') = p$. Moreover, we have $s_p^\sigma = s_p^\sigma$. Further, every action of $p$ enabled at $s_p^\sigma$ should be allowed and $a$ is such an action. So $\tau_1' a \in \text{Run}/\text{Sch}$. Now, we use the fact that $\text{Sch}$ is valid to get a $\tau_3$ with $\tau_1' a \tau_3 \in \text{Run}_{\text{opt}}$. But we have $\tau_1' a \tau_3 \sim \sigma \tau_1 a \tau_3 \sim \sigma \tau_1 a \tau_2$ which contradicts the fact that $\sigma a$ cannot be extended in a complete run.

Suppose $s_p^\sigma$ is a polling state. Let $?c$ be the only action of $p$ enabled at $(s^\sigma, \chi^\sigma)$. That is, $a = ?c$. We thus have $\chi^\sigma(c) > 0$. Thus $\tau = \tau_1 ?c \tau_2$, where $\tau_1$
has no action belonging to \( p \). It means that \( \sigma' \sim \sigma \tau_1 \cap_2 \sigma \tau_1 \tau_2 = \sigma\tau_1 \tau_2 \). This contradicts the fact that \( \sigma \alpha \) cannot be extended into a complete run.

Hence \( Sch_{ca} \) is a valid Schedule for \( A \). \( \Box \)

The concept of an anchored run, that we introduce now will also play a crucial role in what follows. If \( \chi \) is a mapping from \( Ch \) to the non-negative integers, let \( \max(\chi) = \max\{\chi(c) \mid c \in Ch\} \). For a run \( \sigma \), let \( \max(\sigma) = \max\{\max(\chi(\sigma')) \mid \sigma' \) is a prefix of \( \sigma \} \). We say that \( \sigma \) is an anchored run iff \( \max(\sigma) \) is non-null and \( \max(\sigma) > \max(\chi(\sigma')) \) for every strict prefix \( \sigma' \) of \( \sigma \). Anchored runs according to \( Sch_{ca} \) have a special property: every action enabled concurrently with the last action of an anchored run is a send action on some buffer that holds a maximum number of items. This property may be stated precisely as follows.

**Observation 6.** Let \( \sigma \) be an anchored run according to \( Sch_{ca} \), and let \( M = \max(\sigma) \). Then \( \sigma = \sigma c \) for some \( c \in Ch \) and \( \chi(c) = M \). Further, if \( a \in \Sigma \) is enabled at \((s^b, \chi^b)\), then \( a = !d \) for some \( d \in Ch \) and moreover \( \chi^b(d) = M - 1 \).

Notice that \( d = c \) is possible. We are now ready to prove Prop. 3.

**Proof.** of Prop. 3

The if part is obvious. As for the only if part, let \( Sch \) be a valid and regular schedule for \( A \). First, it follows from lemma 5 that \( Sch_{ca} \) is valid.

We prove that \( Sch_{ca} \) is regular. We know that \( RC/Sch \) contains a finite number \( k \) of configurations. Since each action adds at most one item to one buffer, for all \( \sigma \in Run/Sch \), \( \max(\sigma) \leq k \). We will prove that for all \( \sigma_{ca} \in Run/Sch_{ca} \), \( \max(\sigma_{ca}) \leq k \), which will imply that \( RC/Sch_{ca} \) has a finite number of configurations. Since we know that \( Sch_{ca} \) is valid, it suffices to consider only complete runs of \( Run/Sch_{ca} \).

Let \( \sigma_{ca} \in Run/Sch_{ca} \) be a complete run. Following observation 4, let \( \sigma \in Run/Sch \) be a complete run such that \( \sigma \sim \sigma_{ca} \). Suppose \( M_{ca} = \max(\sigma_{ca}) \) and \( M = \max(\sigma) \). Pick the least prefix \( \tau_{ca} \) of \( \sigma_{ca} \) such that \( \tau_{ca} = M_{ca} \). Thus \( \tau_{ca} \) is anchored. By observation 6, let \( \tau_{ca} = \tau_{ca} \cap \). Consider the sequence \( \tau_{ca} \). For a prefix \( \tau \) of \( \sigma \), we say \( \tau \) is covered by \( \tau_{ca} \) iff for every \( p \in P \), \( prj_p(\tau) \) is a prefix of \( prj_p(\tau_{ca}) \). Now pick \( \tau \) to be the least prefix of \( \sigma \) such that \( \tau \) is not covered by \( \tau_{ca} \). Such a \( \tau \) exists, following the definition of \( \sim \). Let \( \tau = \tau a \) where \( a \in \Sigma \) is the last letter of \( \tau \). We consider three cases.

**Case (i)** \( a = !d \) for some \( d \in Ch \).

The choice of \( \tau \) implies \( prj_{p_b}(\tau) = prj_{p_b}(\tau_{ca}) \). Thus, \( s^\tau(p_b) = s^{\tau_{ca}}(p_b) \). And \( !d \) is enabled at configuration \((s^\tau_{ca}, \chi^\tau_{ca})\). It follows from observation 6 that \( \chi^\tau_{ca}(d) = M_{ca} - 1 \) (whether \( d = c \) or not). As \( dst(d) \neq p_b \), the choice of \( \tau \) also implies \( prj_{dst(d)}(\tau) \) is a prefix of \( prj_{dst(d)}(\tau_{ca}) \). Hence, we have \( \#_a(\bar{\tau}) = \#_a(\bar{\tau_{ca}}) \) and \( \#_d(\bar{\tau}) \leq \#_d(\bar{\tau_{ca}}) \), where \( \#_i(p) \) denotes the number of occurrences of letter \( b \) in sequence \( p \). It follows that \( \chi^\tau(d) \geq \chi^{\tau_{ca}}(d) \). Combining these observations with \( \chi^\tau(d) \leq M - 1 \) then yields \( M_{ca} = k \).

**Case (ii)** \( a = ?d \) for some \( d \in Ch \).

By the same argument as in case (i), we have \( s^\tau(p_b) = s^{\tau_{ca}}(p_b) \). Also we have \( prj_{p_b}(\tau) = prj_{p_b}(\tau_{ca}) \), and \( prj_{src(d)}(\tau) \) is a prefix of \( prj_{src(d)}(\tau_{ca}) \). Hence,
\( \chi^\sharp(d) \leq \chi^{\sharp_a} \). It follows that \( ?d \) is enabled at configuration \((s^{\sharp_a}, \chi^{\sharp_a})\). This contradicts that at configuration \((s^{\sharp_a}, \chi^{\sharp_a})\), the schedule \( \text{Sch}_{ca} \) picks process \( src(c) \) with \( s^{\sharp_a}(src(c)) \) being a sending state.

—Case (iii): \( a \in \Sigma_{\text{path}}^a \).

Similar to Case (ii), we obtain a contradiction by noting that \( a \) is enabled at \((s^{\sharp_a}, \chi^{\sharp_a})\).

\[ \square \]

4.2 Deciding Boundedness of the Canonical Schedule.

The decision procedure for boundedness of \( \text{Sch}_{ca} \) is similar to the decision procedure for the boundedness of Petri nets [7], but one cannot directly apply [7] because \( RC/\text{Sch}_{ca} \) cannot be represented as the set of reachable markings of a Petri net. Instead, the canonical schedule performs a zero-test when it schedules a process ready to send, because it must check that all processes ready to receive have empty input buffers. We show that one can nevertheless build a finite tree of configurations in \( RC/\text{Sch}_{ca} \) that exhibits a witness for unboundedness if \( RC/\text{Sch}_{ca} \) is not a finite set or \( \text{Sch}_{ca} \) is not a valid schedule for \( A \).

Towards this, the following partial order relation on anchored runs will play a useful role. Let \( \text{Run}_{an}/\text{Sch}_{ca} \) be the subset of anchored runs of \( \text{Run}/\text{Sch}_{ca} \). We define \( \prec_{ca} \subseteq \text{Run}_{an}/\text{Sch}_{ca} \times \text{Run}_{an}/\text{Sch}_{ca} \) as the least (strict) partial order satisfying the following. For \( \sigma, \sigma' \in \text{Run}_{an}/\text{Sch}_{ca} \), \((\sigma, \sigma')\) is in \( \prec_{ca} \) whenever \( \sigma = \sigma' c, \sigma' = \sigma' ' c \) for some \( c \in Ch \) and:

- \( \sigma \) is a strict prefix of \( \sigma' \).
- \( s^\sigma(p) = s^{\sigma'}(p) \) for every \( p \in P \).
- \( \chi^\sigma(d) \leq \chi^{\sigma'}(d) \) for each \( d \in Ch \).

Notice that, in particular, \( \chi^\sigma(c) < \chi^{\sigma'}(c) \) since \( \sigma \) is a strict prefix of \( \sigma' \) and both are anchored. We show now a structural property of \( \prec_{ca} \) which will serve us to produce a finite coverability tree for all runs. An infinite run of \( A \) is an infinite sequence \( \rho \) in \( \Sigma^\omega \) such that every finite prefix of \( \rho \) is in \( \text{Run}(A) \). We say that an infinite run \( \rho \) is admitted by \( \text{Sch}_{ca} \) iff every finite prefix of \( \rho \) is admitted by \( \text{Sch}_{ca} \).

**Proposition 7.** Suppose \( \rho \in \Sigma^\omega \) is an infinite run admitted by \( \text{Sch}_{ca} \). Then there exist two finite prefixes \( \sigma, \sigma' \) of \( \rho \) such that either \( \sigma, \sigma' \) end at the same configuration, or \( \sigma \prec_{ca} \sigma' \) (in which case \( \sigma, \sigma' \) are both anchored).

**Proof.** If there exists a bound \( k \in \mathbb{N} \) such that for all prefixes \( \alpha \) of \( \rho \), \( \max(\chi^\alpha) \leq k \), there is only a finite number of possible configurations, hence we can find two prefixes of \( \rho \) ending at the same configuration. Else, \( \max(\chi^\alpha) \) is unbounded. It means that we can extract an infinite subsequence of anchored prefixes from the sequence of prefixes of \( \rho \). Since there is a finite number of buffers and a finite number of tuples of local states in \( \Pi_{p \in P}(S_p) \), we can extract an infinite subsequence of anchored prefixes which have the same maximal channel \( c \in Ch \) and the same tuple of local states \( s \in \Pi_{p \in P}(S_p) \).
By an inductive argument on $i \leq |Ch|$, one easily verifies that there are infinitely many anchored prefixes $\alpha_0, \alpha_1, \ldots$ of $\rho$, such that $\chi^{\alpha_0}(c_j) \leq \chi^{\alpha_1}(c_j) \leq \ldots$ for every index $1 \leq j \leq i$. In particular, we get the existence of $\sigma, \sigma'$ with $\sigma \prec_{ca} \sigma'$.

Next we show that any pair of runs $\sigma, \sigma'$ with $\sigma \prec_{ca} \sigma'$ witnesses for the unboundedness of $RC/Sch_{ca}$ (or for the non-validity of $Sch_{ca}$). This requires an argument that differs from [7] because, even though $\sigma' = \sigma$ and both $\sigma, \sigma'$ are according to $Sch_{ca}$, $\sigma \tau^n$ may be incompatible with $Sch_{ca}$ for some $n$ (because of zero-tests). However, we shall argue that if there exist two anchored paths satisfying $\sigma \prec_{ca} \sigma'$ then for every $n = 1, 2, \ldots$, there exists a run $\rho_n$ according to $Sch_{ca}$ such that either $\max(\rho_n) \geq n$ or $\rho_n$ cannot be extended to reach a final configuration.

**Proposition 8.** If there exist two anchored paths $\sigma, \sigma'$ in $Rm_{an}/Sch_{ca}$ such that $\sigma \prec_{ca} \sigma'$, then either $RC/Sch_{ca}$ has an infinite number of configurations or $Sch_{ca}$ is not valid.

**Proof.** Suppose $\sigma' = \sigma \tau$. Fix an arbitrary integer $k > 1$ and consider the sequence $\sigma = \sigma \tau \tau \ldots \tau$ ($k$ copies of $\tau$). Following the definition of $\prec_{ca}$, one verifies that $\tau$ is a run of $A$. If $\alpha$ cannot be extended to a complete run, then $Sch_{ca}$ is not valid and this ends the proof. Else, by observation 4, there exists a complete run $\rho \sim ca \tau$ which is according to $Sch_{ca}$, for some $w \in \Sigma^*$. Let $M = \max(\sigma)$ and $M' = \max(\sigma')$. Let $\sigma = \sigma c; \sigma' = \sigma c'$, where $c \in Ch$, $\chi^\sigma(c) = M$, $\chi^{\sigma'}(c) = M'$. We show below that $\max(\rho) \geq M + k \cdot (M' - M)$ and thus $Sch_{ca}$ is not regular.

Though $\sigma \tau$ is according to $Sch_{ca}$, we note that $\alpha$ is not necessarily a prefix of $\rho$. Let $\alpha = \alpha c$. Consider the sequence $\hat{\alpha}$. For a prefix $\beta$ of $\rho$, we say that $\beta$ is covered by $\hat{\alpha}$ iff for every $p \in \mathcal{P}$, $prj_p(\beta)$ is a prefix of $prj_p(\hat{\alpha})$. Pick $\beta$ to be the least prefix of $\rho$ such that $\beta$ is not covered by $\hat{\alpha}$. Let $\beta = \beta h$ where $h$ is the last letter of $\beta$. Let $p_h \in \mathcal{P}$ be the process such that $h \in \Sigma_{ph}$. The choice of $\beta$ implies that $prj_{p_h}(\hat{\beta}) = prj_{p_h}(\hat{\alpha})$, and thus $s^\beta(p_h) = s^\alpha(p_h)$. Again we consider three cases.

--- Case (i). $b = \beta d$ for some $d \in Ch$.

Thus, $\beta d$ is enabled at configuration $(s^\beta, \chi^\beta)$. Also, as $dst(d) \neq p_h$, we have that $prj_{dst(d)}(\hat{\beta})$ is a prefix of $prj_{dst(d)}(\hat{\alpha})$. Thus, we have $\#_{\beta d}(\hat{\beta}) = \#_{\beta d}(\hat{\alpha})$, and $\#_{\beta d}(\hat{\beta}) \leq \#_{\beta d}(\hat{\alpha})$, where $\#_a(\theta)$ denotes the number of occurrences of letter $a$ in sequence $\theta$. It follows that $\chi^{\beta d}(d) \geq \chi^\beta(d)$.

Note that $\chi^{\beta d}(c) = M + k \cdot (M' - M) - 1$ and $\chi^{\beta d}(d) \leq \max(\rho) - 1$. Thus, if $d = c$, then we have $\max(\rho) \geq M + k \cdot (M' - M)$. Otherwise, $d \neq c$. By definition of $\prec_{ca}$, we conclude that $\beta d$ is also enabled at both configurations $\chi^\beta(c), (s^\beta, \chi^\beta)$. Thus, we have $\chi^{\beta d}(d) = M - 1, \chi^{\beta d}(M' - M) = \chi^{\beta d}(d) - 1$, due to observation 6. It follows that $\chi^{\beta d}(M' - M) = \chi^{\beta d}(d) - 1$. Consequently, we also have $\max(\rho) = M + k \cdot (M' - M)$.

--- Case (ii). $b = \beta d$ for some $d \in Ch$.

Following the definition of $\prec_{ca}$, we have $s^\beta(p_h) = s^{\beta d}(p_h) = s^\alpha(p_h) = s^\beta(p_h)$. At configuration $(s^\beta, \chi^\beta)$, $Sch_{ca}$ picks process $src(c)$ where $s^\beta(src(c))$ is a sending
state. Hence, \( p_b \) is not enabled at \((s^\delta, \chi^\delta)\). That is, \( \chi^\delta(d) = 0 \). Similarly, we have \( \chi^\delta(d) = 0 \). As a result, \( \chi^\delta(d) = 0 \).

However, by similar arguments as in case (i), one sees that \( \#_d(\hat{\beta}) = \#_d(\hat{\alpha}) \) and \( \#_d(\hat{\beta}) \leq \#_d(\hat{\alpha}) \). Thus, \( \chi^\beta(d) \leq \chi^\alpha(d) \). We obtain a contradiction as \( \hat{\alpha} \) is enabled at configuration \((s^\beta, \chi^\beta)\).

---

**Case (iii).** \( b \in \Sigma_{\text{cho}} \).

Similar to Case (ii), we obtain a contradiction by noting that \( p_b \) is enabled at \((s^\delta, \chi^\delta)\). \( \square \)

The set of all runs of a data-branching system under the canonical schedule \( \text{Sch}_{ca} \) forms a possibly infinite tree (any data-dependent choice performed by a scheduled process induces several branches). Following Karp and Miller’s ideas, one may stop exploring this tree whenever coming again to a configuration already visited, or obtaining an anchored run \( \sigma' \) that extends a smaller anchored run \( \sigma \), i.e. \( \sigma \prec_{ca} \sigma' \). Based on this construction of a finite coverability tree, we obtain the following theorem.

**Theorem 9.** One can effectively determine whether \( \text{Sch}_{ca} \) is valid and regular.

**Proof.** We construct inductively \( W \), a tree of valid runs admitted by \( \text{Sch}_{ca} \). First, \( \varepsilon \) is in \( W \). Then, suppose that \( \sigma \) is in \( W \) and \( \sigma \alpha \) is a run admitted by \( \text{Sch}_{ca} \), where \( \alpha \in \Sigma \). If there exists \( \sigma' \in W \) such that \( \sigma' \prec_{ca} \sigma \alpha \), then by proposition 8, we can stop the construction of \( W \) and report that either \( \text{Sch}_{ca} \) is not regular or \( \text{Sch}_{ca} \) is not valid. Otherwise, we check if there exists \( \tau \in W \) such that \( \tau \) ends at the same configuration as \( \sigma \alpha \). If such a \( \tau \) does not exist, then we add \( \sigma \alpha \) to \( W \) (otherwise we just ignore \( \sigma \alpha \)).

We first prove that the construction of \( W \) stops after a finite number of steps. Suppose otherwise. Then members of \( W \) form an infinite tree. By König’s lemma, there exists an infinite sequence \( \rho \) of \( \Sigma^* \) such that every finite prefix of \( \rho \) is in \( W \). Applying proposition 7, we get that there exist two finite prefixes \( \sigma, \sigma' \) of \( \rho \) such that \( \sigma \) is a prefix of \( \sigma' \) and either \( \sigma, \sigma' \) end at the same configuration or \( \sigma \prec_{ca} \sigma' \). In both cases, the construction would not extend \( \sigma' \), hence \( \rho \) is not an infinite path, a contradiction.

If the above construction of \( W \) terminates without finding \( \sigma \prec_{ca} \sigma' \) (reporting that \( \text{Sch}_{ca} \) is not regular or that \( \text{Sch}_{ca} \) is not valid), then \( \{(s^\sigma, \chi^\sigma) \mid \sigma \in W\} \) is exactly the set of configurations of \( \text{Sch}_{ca}(\text{RC}) \), that is we have the proof that \( \text{RC}/\text{Sch}_{ca} \) is a finite set, and we can test easily whether \( \text{Sch}_{ca} \) is valid. \( \square \)

---

![Fig. 12. A data-branching system.](image-url)
Thm. 2 is now settled by applying Prop. 3 and Thm. 9.

To illustrate the construction in Thm. 9, we consider the data-branching system in Fig. 12 and display in Fig. 13 the corresponding tree $W$ of Thm. 9. In Fig. 13, the root is indicated by a pointing arrow and each node $\mathcal{N} \sigma$ is identified by reading the labels along the path from the root to $\mathcal{N} \sigma$. We label each node $\sigma$ with the configuration at which $\sigma$ ends, where the notation $i j k l m$ represents the configuration in which process $P2$ is at state $i$, $P1$ is at state $j$, buffer $c,e,o$ have respectively sizes $k,l,m$. A dotted arrow represents a branching which leads to a node already constructed.

First, $\mathcal{S} \mathcal{C}_{ca}$ succeeds to reach the final state 5E000 through the path $af b c ? e g ! e ? e o ? o$ (we compressed the fourth last transitions as one dashed transition since no choice actions are involved). Two deadend states are reached: 2D010 and 5E100. It means that $\mathcal{S} \mathcal{C}_{ca}$ is not valid, hence applying Prop. 3, we know that no valid schedule exists for $\mathcal{A}$. Finally, the algorithm finds two anchored runs ordered by $\prec$ and reachable from each other, namely $ag ! e \prec_{ca} ag ! e ! c ! e$ (notice that the runs $ag$ and $af$ are not anchored as $\max(a g) = \max(a f) = 0$, hence for instance we do not have $ag \prec_{ca} a g ! e ! e$). The algorithm thus stops at the configuration 3A200 reached by $ag ! e ! e ! e$, and depicted by a double circle around the configuration. It means that the canonical schedule is not regular, and thus applying Prop. 3, we know that no regular and valid schedule exists for $\mathcal{A}$. Recall that the model $\mathcal{A}$ is only an over-approximation of the real system. That is, even if we have a proof that $\mathcal{A}$ is not schedulable, it does not imply that the real system is not schedulable.

Fig. 13. The tree $W$ in Thm. 9 for the system in Fig. 12.
The algorithm gives counterexample runs (here, we have two runs for dead-
ends and one for unboundedness). One can check whether these runs are concrete
runs from the original systems or only spurious runs created by the abstractions.
If they are spurious runs, then the real system is schedulable.

5 Discussion

In this paper, we have considered quasi-static scheduling as introduced in [3] and
have provided a negative answer to an open question posed in [8]. Specifically we
have shown that for the chosen class of infinite state systems, checking whether
a system is quasi-static schedulable is undecidable. We have then identified the
data-branching restriction, and proved that the quasi-static scheduling problem
is decidable for data-branching systems. Further, our proof constructs both the
schedule and the finite state behaviour of the system under schedule. An im-
portant concept used in the proof is the canonical schedule that draws much
inspiration from the study of existential bounds on channels of communicating
systems [5]. In the language of [5], our result can be rephrased as: it is deci-
able whether a weak FIFO data branching communicating system is existentially
bounded, when all its local final states are polling states. We recall that the same
problem is undecidable [5] for strong FIFO communicating systems, even if they
are deterministic and deadend free. Our abstraction policy is similar to the one
used in [10]. However, we use existential boundedness while [10] checks whether
a communicating system is universally bounded, which is an easier notion to
check. Note that the canonical schedule may be easily realized in any practical
context: it suffices to prevent any process from sending to a buffer that already
contains the maximum number of items determined from that schedule. It is
worth recalling that these bounds are optimal.

Deadends play an important role in the notion of quasi-static schedulability
studied here and previously. However, quasi-static scheduling may stumble on
spurious deadends due to the modelling of the task code by an abstract system.
The algorithm we have sketched for constructing the canonical schedule may
be combined with an iterative removal of spurious deadends. A more ambitious
extension would be to accommodate non data-branching systems. For this purpose,
it would be interesting to formulate a notion of quasi-static schedulability based
purely on existential boundedness and to study decidability issues in this setting.

References

static scheduling of independent tasks for reactive systems. IEEE Trans. on


