Regular Specifications of Resource Requirements for Embedded Control Software

Rajeev Alur and Gera Weiss
University of Pennsylvania

Abstract

For embedded control systems, a schedule for the allocation of resources to a software component can be described by an infinite word whose ith symbol models the resources used at the ith sampling interval. Dependency of performance on schedules can be formally modeled by an automaton (ω-regular language) which captures all the schedules that keep the system within performance requirements. We show how such an automaton is constructed for linear control designs and exponential stability or settling time performance requirements. Then, we explore the use of the automaton for online scheduling and for schedulability analysis. As a case study, we examine how this approach can be applied for the LQG control design. We demonstrate, by examples, that online schedulers can be used to guarantee performance in worst-case condition together with good performance in normal conditions. We also provide examples of schedulability analysis.

1 Introduction

A key question in the design and implementation of embedded real-time systems is: *how does one specify the resource requirements of a component?* When the resource is CPU, the most commonly used framework for specifying the usage requirements is the *periodic task* model [12]: each component specifies a period, sometimes along with a deadline, which gives the frequency at which the component must execute. The designer of the component makes sure that the performance objectives will be met as long as the component is executed consistent with its period. For implementation, the real-time operating system performs a worst-case execution time analysis on all the components, followed by schedulability analysis to check whether all the timing requirements can be met (c.f. [10, 4]).

Specifying resource requirements using periods has advantages due to simplicity and analyzability, but has some key deficiencies. First, periodic task model has limited expressiveness: a specification such as "execute the component every 5ms" does not say whether the scheduler should or should not execute it more frequently if enough computing resources are available. For some tasks such as updating sensor readings of environment, as the load decreases, the component should be executed more frequently and this will improve the system performance. For some tasks such as refreshing of display, a fixed period is reasonable. For control systems, while periodic task specifications (e.g., "once every 5 slots") capture only the worst-case bounds, we cannot simply treat these as upper bounds, and use "at least once every 5 slots" as the specification. This is consistent with the common wisdom in control theory that more frequent execution need not imply better performance. In general, for complex systems consisting of a mix of tasks, the framework should allow flexible and expressive specifications of resource requirements. Second, such specifications do not compose in the sense that a system composed of two components cannot be specified by a single period, and when a component is added the schedulability analysis must be performed again on the global set of tasks.

The focus of this paper is performance of the control system. Our goal is to guarantee that the system meets its performance requirements. We take a control design with performance objective such as exponential stability or settling time, and compute a specification of resource requirement. We use formal languages and finite automata over infinite words as an expressive, analyzable, and composable specification framework for resource requirements. We assume that resources are allocated in discrete slots of some fixed duration. Such a virtual time-triggered allocation strategy may be because of time-triggered architecture [9] or because the system supports the FLET (Fixed Logical Execution Time) programming abstraction [8]. Given a component, the allocation of the resource to that component in a particular execution can be described by an infinite word $\sigma = \sigma_1\sigma_2\cdots$, where each symbol σ_i describes whether the component was scheduled during the ith slot or not. The resource requirement of a component, then, can be specified by a language L of infinite words that describes all acceptable schedules. The designer of the component makes sure that the performance objectives will be met on all schedules

in L, and the scheduler must ensure that the runtime allocation of the resource to the component corresponds to some word in L. We will assume that L is specified by finite automata with acceptance conditions for infinite words (such as Büchi automata), or equivalently, by ω-regular expressions (see [21] for an introduction to theory of languages of infinite words). In the literature, one can find other formalisms for expressing dependencies between tasks and execution constraints [2, 3, 7, 17]. We use automata, because it can express the semantics of all these models and because we are interested in control performance, not efficient input languages.

We consider two ways of formalizing and composing specifications. In the first, we assume that the resource can be allocated to only one component in any given slot. In this case, the specification of a component with task identifiers I, is a language over the alphabet $I \cup \{0\}$, where $0 \notin I$ means that the slot is not allocated to this component. The composition of two specifications can be computed by an appropriate product construction on automata, and test of schedulability corresponds to testing of language emptiness. In the second style of specification, we assume that multiple tasks can be allocated in a given slot. Then, the specification of a component containing tasks from a set I of identifiers is an ω-language over the alphabet 2^I: the ith symbol of the word gives the set of tasks scheduled in the ith slot (the empty set means that none of the tasks of this component are scheduled in this slot). Composing two specifications L_1 and L_2 over task identifiers I_1 and I_2, yields a language L over the task set $I_1 \cup I_2$ obtained using an appropriate product construction. This more general style of requirements gives platform-independent specifications. On a pacific platform P, we first need to compute the *feasible* subsets of I that can be scheduled in a slot (this can be done using worst-case execution time analysis). The platform-specific specification is then computed by intersection of L with the set of all feasible schedules on P, and tested for emptiness to determine schedulability.

To obtain the above specifications, we offer a methodology for designing control systems, as follows. First, a feedback law is designed for each subset of resources that may become available at some time. For example, if only one resource is to be scheduled, we design two feedback laws F_0 and F_1 for times where the resource is not used and for time where it is used, respectively. Then, we identify the functions f_0 and f_1 that map the state of the closed-loop system (plant and controller) to the next state when F_0 and F_1 are applied, respectively. This gives us a switched system $x(t+1) = f_{\sigma(t)}(x(t))$ where $\sigma\colon \mathbb{N} \to \{0,1\}$ is such that $\sigma(t) = 1$ if and only if the resource is used at time t. For stability requirements, in the case that f_0 and f_1 are linear, we give an algorithm that extracts an automaton for the switched system. This automaton is such that the switched system is exponentially sable for every schedule in its ω-language. This automaton can be used to combine the subsystem with other software components as described above.

The theoretical part is complemented with detailed examples that demonstrate how the framework is applied to particular case studies. The focus of the examples is scheduling CPU resources for an LQG controller. This controller takes the form of two layers. One layers is a standard control design expressed as linear transfer functions. The other layer is an heavy computation that feeds information to the control design. Assuming that updating the linear transfer functions takes negligible computational resources, we design two modes. The first mode is the original control design and the second is a control design that does not use data from the heavy computation. We detail the computation of an automaton for this particular system. Using that automaton we show, by a simulation experiment, how the expressiveness of the automata interface allows a scheduler to take advantage of unused slots and, by that, improve controller performance. We also demonstrate how automata based interface allows schedulability analysis that leads to refinement of control objectives, until an acceptable schedule is found. When the system becomes non schedulable, we demonstrate the use of the automata interface in constructing an platform-independent description of the scheduling constraints that can be used to choose an implementation platform. The types of performance requirements considered are exponential stability and settling-time of the step response. The examples are computed using a prototype implementation of our algorithms. This implementation is a Mathematica notebook that uses built-in and external tools. The amount of time needed to compute the examples with our prototype implementation are reported for reference.

Related Work

Many researchers have identified the lack of composability as a problem for scalable component-based design and integration, and offered composable and hierarchical scheduling frameworks [6, 18, 14, 19]. For example, [19] proposes the periodic resource model, where the specification of a component consists of (T, C) meaning that the component should get C units of computation every T units of time, and shows how to abstract a set of periodic tasks with EDF or rate-monotonic scheduling policies into a single periodic resource. While these efforts address composability, the expressiveness is still limited to specifying periods for individual components.

Formal methods literature consists of general frameworks such as I/O automata [13], fair transition systems [15], and interface automata [5] for capturing inter-

faces with well-developed theories of composition and refinement. Our use of automata is consistent with such general frameworks, and can be viewed as "light-weight" instantiation for the specific purpose of scheduling. Timed automata have also been used for schedulability analysis [1]. The idea of using formal languages and Büchi automata as an interface to capture the set of acceptable schedules over the alphabet of task identifiers, was first advocated in our recent paper [23], which focuses on specifying stability of switched systems using automata. In the current paper we use the proposed concepts to instantiate a methodology for control systems design. To this end, we establish the following additions. First, expressibility is enhanced by allowing multiple tasks per slot. Second, we show how a switched system is obtained for linear control designs. Thirds, detailed case studies show how the resulting automaton can be used for practical applications.

2 Automata based specifications

2.1 ω-regular languages

We review the basic definitions related to ω-automata and ω-languages [21, 22]. Given an alphabet Σ, an ω-word is an infinite sequence $\sigma_1\sigma_2\ldots$ with each $\sigma_i \in \Sigma$. An ω-language L over Σ is a set of ω-words. A Büchi automaton A over Σ consists of a finite set of states Q, an initial state $q_0 \in Q$, a transition function $\delta : Q \times \Sigma \mapsto 2^Q$, and a set $F \subseteq Q$ of accepting states. A run of the automaton A over an ω-word $\sigma_1\sigma_2\ldots$ consists of an infinite sequence of states $q_0q_1q_2\ldots$ starting at the initial state such that $q_i \in \delta(q_{i-1}, \sigma_i)$ for each $i > 0$. The run is accepting iff for infinitely many positions i, $q_i \in F$. The language $L(A)$ of the automaton consists of ω-words σ such that A has an accepting run over σ. An ω-language L is said to be ω-regular iff there is a Büchi automaton A such that $L(A) = L$.

A Büchi automaton $A = (Q, q_0, \delta, F)$ is *deterministic* if for all states q and symbols a, $|\delta(q, a)| \leq 1$. A deterministic automaton has at most one run over a given ω-word. A Büchi automaton $A = (Q, q_0, \delta, F)$ is a *safety automaton* if $F = Q$, that is, a word is accepted as long as there is an infinite run.

The ω-regular languages are effectively closed under a variety of operations such as language intersection and language homomorphisms. To check whether the language $L(A)$ of a Büchi automaton is non-empty, it suffices to consider the transition graph G_A of the automaton: the nodes in G_A are states of A, and there is an edge from q to q' iff $q' \in \delta(q, a)$ for some symbol a. The language $L(A)$ is non-empty iff there is a cycle in G_A that is reachable from the initial state q_0 and contains some state in F. The automaton A is said to be *trim* if for every state q, there is a cycle in G_A that is reachable from q and contains some state in F. Every automaton with non-empty language can be converted into an equivalent trim one by deleting redundant states and transitions.

2.2 Exclusive slot allocation

Suppose each slot can be allocated to at most one task. Let I be the set of task identifiers of all the tasks in a component. Then, from the point of view of a component, a schedule can be represented by an infinite sequence $w = \sigma_1\sigma_2\ldots$ over task identifiers I, along with a special symbol $0 \notin I$: for each slot i, $\sigma_i = 0$ if the slot is not allocated to the component, and otherwise σ_i specifies the task that was allocated the ith slot. The *specification* S of a component is (I, L), where I is the set of task identifiers with $0 \notin I$, and L is ω-language over the alphabet $I \cup \{0\}$.

Note that the specification exposes the set of task identifiers within a component, but this information is necessary for the scheduler to make resource allocation.

Consider two components whose resource requirements are specified as (I_1, L_1) and (I_2, L_2). Typically, I_1 and I_2 will be disjoint, but this is not required. In fact, one can have have $I_1 = I_2$, and in this case, L_1 and L_2 are specifying two distinct requirements on scheduling such that the composition corresponds to conjoining the requirements. Suppose the ω-languages L_1 and L_2 are given by automata $A_1 = (Q_1, q_0^1, \delta_1, F_1)$ and $A_2 = (Q_2, q_0^2, \delta_2, F_2)$, respectively. The composed specification is $(I_1 \cup I_2, L)$, where the composed language L can be computed using a modified product construction. Let us first assume that the automata are safety automata. The composed automaton A has states $Q_1 \times Q_2$ with initial state (q_0^1, q_0^2). The transition relation δ is specified as follows: For $a = 0$ and for $a \in I_1 \cap I_2$, $(p_1, p_2) \in \delta((q_1, q_2), a)$ iff $p_1 \in \delta_1(q_1, a)$ and $p_2 \in \delta_2(q_2, a)$; for $a \in I_1 \setminus I_2$, $(p_1, p_2) \in \delta((q_1, q_2), a)$ iff $p_1 \in \delta_1(q_1, a)$ and $p_2 \in \delta_2(q_2, 0)$; and analogously, for $a \in I_2 \setminus I_1$, $(p_1, p_2) \in \delta((q_1, q_2), a)$ iff $p_1 \in \delta_1(q_1, 0)$ and $p_2 \in \delta_2(q_2, a)$. Thus, for a task a belonging to only one of the components, the a-transitions of this component are synchronized with 0-transitions of the other component. If the automata have non-trivial accepting conditions, then we need to add a bit to the product states to make sure that accepting states of both are visited infinitely often as in the classical intersection of Büchi automata [22].

After constructing the product, we apply the trimming operation to get rid of redundant states. If the result of trimming is the empty automaton, then the two components cannot be composed due to scheduling conflicts.

2.3 Scheduling task sets per slot

While the assumption of one task per slot gives useful specifications, for better utilization of resources, the specification can be made richer. For a component with set I of tasks, resource allocation from the point of view of this component is now specified by an ω-word $w = \sigma_1\sigma_2\ldots$ such that each $\sigma_i \subseteq I$ gives all the tasks of this component that are scheduled during the ith slot. If σ_i is the empty set, then no task of this component is scheduled during this slot. The resource requirement of the component is given as the specification (I, L), where L is an ω-language over the alphabet 2^I of task sets.

A direct motivation for using this type of specification is for scheduling sets of identical resources. For example, if we want to schedule tasks to run on a dual processor system, we may allow two tasks per slot. Another motivation is as follows. The FLET (Fixed Logical Execution Time) programming abstraction [8] is a useful methodology for designing control software. The main idea in FLET is fixing the times where data is moved between internal software variables and actuator/sensors. The standard use of this approach is to choose a set of tasks to run between updates and make sure that they are schedulable within the given time. We propose using automata to schedule possibly different task sets to slots. In addition to improved performance, this approach allows platform independent analysis that can be combined with platform-dependent information to choose a platform (see Section 4.3 below for examples).

Consider two components with specifications (I_1, L_1) and (I_2, L_2). Suppose the Büchi automata A_1 and A_2 represent their specifications. Then the product automaton A representing the composite specification over the task set $I_1 \cup I_2$ is computed as follows. The state-space is $Q_1 \times Q_2$ with initial state (q_0^1, q_0^2). The transition relation δ is specified by: for a task set $\alpha \subseteq I_1 \cup I_2$, $(p_1, p_2) \in \delta((q_1, q_2), \alpha)$ iff $p_1 \in \delta_1(q_1, \alpha \cap I_1)$ and $p_2 \in \delta_2(q_2, \alpha \cap I_2)$.

For schedulability analysis, given a component with task set I, we must first determine, for each set $\alpha \subseteq I$, if α is *feasible*: during one slot, can all tasks in α be executed. This requires estimates of the execution times of all the tasks in I. Formally, a *platform* P is a mapping from 2^I to $\{0, 1\}$ indicating feasibility of each task set. A schedule $\sigma_1\sigma_2\ldots$ is feasible on a platform P if $P(\sigma_i) = 1$ for each i. Let the set of all feasible schedules on a platform P be L_P. Given a component specification $S = (I, L)$ and a platform $P : 2^I \mapsto \{0, 1\}$, the platform-specific specification is $(I, L \cap L_P)$. Given an automaton A over 2^I representing the resource specification L, we can obtain the feasible schedules in L by deleting all transitions corresponding to infeasible task sets, and trimming the automaton. If the resulting language $L \cap L_P$ is empty, then the component is not schedulable on the platform P.

Note that, with this approach, the same specification can be used for different platforms. Also, the exclusive slot allocation discussed in Section 2.2 is a special case of a platform where only singleton task sets are feasible.

3 Scheduling resources for control software

In this section we outline a methodology for designing resource usage schedulers in software based controllers (software which monitors and affects a physical plants).

The proposed methodology relies on the analysis exposed in [23], where the synthesis of schedules (switching-signals) for switched systems is discussed. Here, we show how this framework can be used to schedule resources in control software. Specifically, we show how scheduling usage of resources in control software translates to finding a switching signal in a switched system. We also show how scheduling automata can be used to improve the design of software controllers. Automata based scheduling is best suited for scheduling computational resources, where it is critical that the scheduler itself does not consume significant computational resources.

Software control systems typically operate at a fixed frequency. After reading each new sample from the sensor, the software reacts to the plant's changed output by recalculating and adjusting the drive signal. The plant responds to this change, another sample is taken, and the cycle repeats. Eventually, the plant should reach the desired state and the software will cease making changes. When disturbances arrive or a new setpoint is chosen, the process repeats itself.

We assume that a linear model, which is an approximation of the dynamics of a physical plant, is given. Specifically, consider a plant modeled as a, so called, discrete-time linear time-invariant system (see e.g. [20])

$$\begin{aligned} x_p(t+1) &= A_p x_p(t) + B_p u(t) \\ y(t) &= C_p x_p(t) \end{aligned}$$

consisting of state equation, defined in terms of the matrices A_p and B_p, and output equation that maps the state to the output, defined in terms of C_p. The state is denoted by x_p, the output by y, and the control input by u.

A feedback control software for this plant takes the output y and decides what action should be taken by the manipulated variable u to remove errors. Many controller designs take the form of a dynamic feedback, that is, the controller is a dynamical system by itself. More specifically, the controller is designed as a linear time-invariant system such that, when the output of the controller is attached to the input of the system and vice verse, the composed system is stable. Mathematically, assume that the controller is

described by the discrete-time linear time-invariant system

$$\begin{aligned} x_c(t+1) &= A_c x_c(t) + B_c y(t) \\ u(t) &= C_c x_c(t) \end{aligned}$$

where x_c is the state of the controller and A_c,B_c and C_c are, respectively, the state transition matrix, the input map, and the output map for the controller. Then, it is easy to verify that the dynamics of the composed system (the controller and the plant together) can be described by

$$x(t+1) = \begin{pmatrix} A_p & B_p C_c \\ B_c C_p & A_c \end{pmatrix} x(t)$$

where $x = (x_p^T, x_c^T)^T$ is the concatenation of the states of the plant and the controller.

To model limitation of resources, we use switched systems [11]. More specifically, assume that access to certain input and/or output variables requires allocation of shared resources. By rewiring the B_c matrix, for output variables, and C_c matrix, for input variables, we model resource allocation as follows. A switched system is defined with a mode for every subset of resources that may be available at a time. The modes that correspond to not using the ith input or output variable have zero at the ith entry of the C_c or B_c matrix, respectively. The A_c matrices are designed accordingly. With this abstraction, switching sequences for the resulting switched system correspond to allocation of resources to the control loop. Let m be the number of modes. For every mode $i = 1, \ldots, m$ we have the controller

$$\begin{aligned} x_c(t+1) &= A_{c_i} x_c(t) + B_{c_i} y(t) \\ u(t) &= C_{c_i} x_c(t). \end{aligned}$$

The composition of the system with the controller modes gives the closed-loop switched system

$$x(t+1) = A_{\sigma_t} x(t) \tag{1}$$

where $\sigma = \sigma_1 \sigma_2 \ldots$ is such that σ_t reflects the resources used at time t and

$$A_i = \begin{pmatrix} A_p & B_p C_{c_i} \\ B_{c_i} C_p & A_{c_i} \end{pmatrix}, \ i = 1, \ldots, m.$$

The next step is an analysis of the system (1). We use formal-languages techniques to characterize the set of schedules that meet performance specifications. Specifically, an automaton is computed whose language is the set of schedules that guarantee exponential stability.

For the parameters $l \in \mathbb{N}$ and $\rho \in (0, 1]$, a system is said to be (l, ρ)-exponentially-stable if $\|x(t+l)\|/\|x(t)\| < \rho$ for every $t \in \mathbb{N}$ and $x(t) \in \mathbb{R}^n$. For the system (1), consider the language

$$\begin{aligned} G_{l,\rho} &= \{\sigma : \frac{\|x(t+l)\|}{\|x(t)\|} < \rho \text{ for all } t \text{ and } x(t)\} \\ &= \{\sigma : \|A_{\sigma_{t+l}} \cdots A_{\sigma_{t+1}}\| < \rho \text{ for all } t \text{ and } x(t)\}. \end{aligned}$$

The equivalence follows from the definition of matrix norm.

Given $l \in \mathbb{N}$ and $\rho \in (0, 1]$, we propose the following algorithm: (1) Construct the set $B = \{\sigma : \|A_{\sigma_l} \cdots A_{\sigma_1}\| \geq \rho\}$. (2) Build the regular expression $\{1, \ldots, m\}^* - \sum_{\sigma \in B} \{1, \ldots, m\}^* \sigma \{1, \ldots, m\}^*$. (3) Translate the regular expression to a deterministic finite automaton. (4) Delete all states from which there is no path to an accepting state.

Proposition 3.1. *The above algorithm computes a Büchi automaton for $G_{l,\rho}$.*

Proof. The set B contains all the words of length l that are not allowed as subwords. For a word $\sigma \in B$, the sub-expression $\{1, \ldots, m\}^* \sigma \{1, \ldots, m\}^*$ defines the language of words that contain σ as a subword. The sum $\sum_{\sigma \in B} \{1, \ldots, m\}^* \sigma \{1, \ldots, m\}^*$ is the language of all words that contain a word in B as a subword. Therefor, the whole expression defines the set of words that do not contain any of the words in B as a subword.

Next, we argue that same conclusion remains if we consider the automaton as a Büchi automaton. Because all non accepting states are traps, an infinite run is accepted iff all the states are accepting. Since this is also the accepting condition for finite words, the conclusion that the automaton accepts all words that do not have a word in B as a subword remains valid also for infinite words.

When infinite words are concerned, a word can be rejected once the automaton gets to a state from which there is not path to an accepting state. Thus, deleting such states does not change the language of the automaton. □

The resulting automaton is a description of all schedules in $G_{l,\rho}$. When the requirement for the system is exponential stability, it is a finite representation of all acceptable schedules. It can be used both for schedulability analysis and as a practical tool for online scheduling, as follows.

For schedulability analysis, assume that we have computed automata for all the subsystems that share a resource. Then, using algorithms for automata intersection, we can compute an automaton for the language of all schedules that meet the requirements of all subsystems. If this language is not empty, the system is schedulable. Having a representation of all the schedules that satisfy the specifications allows the selection of a good schedule based on optimization criteria. See the next section for examples of using automata for schedulability analysis.

For online scheduling, the automaton can be used as an effective decision procedure. Following the scheduled tasks and updating the current state of the automaton should not take significant computational resources. Since the automaton represents the set of allowed schedules, it can be used to infer the set of tasks that can be scheduled in every computation slot. If, at each slot, we only choose a task that labels an edge that exists the current state of the automaton, we

are guaranteed that we will be able to do this forever and that the resulting schedule is acceptable for all subsystems. Examples for using automata as online schedulers are given in the next section.

4 Case studies in control and scheduling

In this section we explore applications of the proposed methodology. We give an explicit controller design with two operating modes - one that uses heavy computational resources and one that only use lightweight computations. Using this example, we show how automata can be used for both schedulability analysis and online scheduling.

4.1 A controller design

Consider a system where reading the plant output is computationally demanding (e.g. an heavy image processing algorithm is needed for obtaining the output). We propose two modes of operation for the controller – one that uses the output measurement and one that does not use it. Then, schedules of invocation of the heavy algorithm correspond to mode switches of the control system. The controller itself is built of blocks with linear dynamics that can be computed with negligible resources.

Linear quadratic Gaussian (LQG) control is a standard method of designing feedback control laws for linear systems with additive Gaussian noise. We examine how this method can be adopted to operate when there are not enough computational resources to get the output in every sampling interval.

An LQG controller is a combination of the solutions of estimator and full-state feedback, based on the so-called separation principle. It can be automatically computed using the `lqg` command in MATLAB [24]. The structure of the controller is depicted in Figure 1.

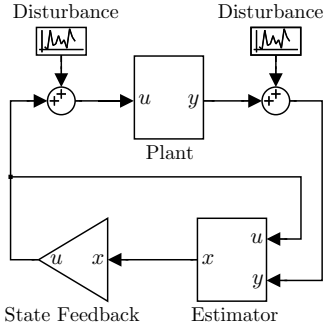

Figure 1. Closed loop with LQG feedback.

If the output can be computed in every slot, we can use this design as is and get best performance. Otherwise, we need to specify how the control signal is produced when the output is not available.

The second mode of the controller, that operates when the output cannot be computed, carry a simulation of the plant. As seen in Figure 2, the structure is very similar to that of the LQG controller. The difference being that a simulation block replaces the estimation block and the output of the plant is not used.

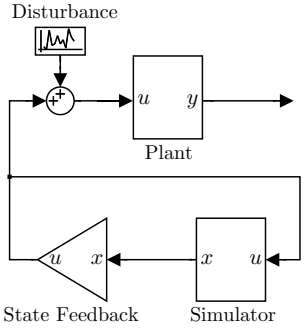

Figure 2. Simulation mode.

The dynamics of the simulation blocks are the dynamics of the system. The idea is to use the last estimation as a starting point and then simulate the dynamics of the system, assuming no noise. Similar ideas have been proposed in networked control systems (see e.g., [16]).

Formally, let

$$\begin{aligned} x_p(t+1) &= A_p x_p(t) + B_p u(t) \\ y(t) &= C_p x_p(t) \end{aligned}$$

be a linear time-invariant model of the plant and

$$\begin{aligned} x_c(t+1) &= A_{c_1} x_c(t) + B_{c_1}(t) y(t) \\ u(t) &= C_{c_1} x_c(t) \end{aligned}$$

a linear-time invariant model for the LQG controller.

The simulation mode, depicted in Figure 2, is formally modeled by

$$\begin{aligned} x_c(t+1) &= A_{c_0} x_c(t) + B_{c_0} y(t) \\ u(t) &= C_{c_0} x_c(t) \end{aligned}$$

where $A_{c_0} = A_p + B_p C_{c_1}$, $B_{c_0} = 0$ and $C_{c_0} = C_{c_1}$.

The composition of the system with the controller modes gives the closed-loop switched system

$$x(t+1) = A_{\sigma_t} x(t) \tag{2}$$

where

$$A_0 = \begin{pmatrix} A_p & B_p C_{c_1} \\ 0 & A_p + B_{c_1} C_p \end{pmatrix}, A_1 = \begin{pmatrix} A_p & B_p C_{c_1} \\ B_{c_1} C_p & A_{c_1} \end{pmatrix}.$$

and $\sigma = \sigma_1\sigma_2\cdots \in \{0,1\}^\omega$ is such that $\sigma_t = 1$ iff the output $y(t)$ is available to the controller (the computation that evaluates y is scheduled).

As a specific example, consider the plant

$$\begin{aligned} \dot{x}_p &= \begin{pmatrix} -1 & 1 \\ 1 & -1 \end{pmatrix} x_p + \begin{pmatrix} 1 \\ 0 \end{pmatrix} u \\ y &= (0,1)x_p. \end{aligned}$$

Sampling every time unit gives the discrete-time matrices:

$$\begin{aligned} A_p &= \begin{pmatrix} \frac{1+e^2}{2e^2} & \frac{-1+e^2}{2e^2} \\ \frac{-1+e^2}{2e^2} & \frac{1+e^2}{2e^2} \end{pmatrix}, \\ B_p &= \begin{pmatrix} \frac{3}{4} - \frac{1}{4e^2} \\ \frac{1}{4}\left(1+\frac{1}{e^2}\right) \end{pmatrix}, \\ C_p &= (0,1). \end{aligned}$$

The `lqg` MATLAB command gives

$$\begin{aligned} A_{c_1} &= \begin{pmatrix} 0.21029 & -0.31865 \\ 0.29069 & -0.027978 \end{pmatrix}, \\ B_{c_1} &= \begin{pmatrix} 0.41233 \\ 0.46143 \end{pmatrix}, \\ C_{c_1} &= (-0.49902, -0.47288). \end{aligned}$$

The closed loop matrices are

$$\begin{aligned} A_1 &= \begin{pmatrix} A_p & B_pC_{c_1} \\ B_{c_1}C_p & A_c \end{pmatrix} \\ &= \begin{pmatrix} 0.568 & 0.432 & -0.357 & -0.339 \\ 0.432 & 0.568 & -0.142 & -0.134 \\ 0 & 0.412 & 0.210 & -0.319 \\ 0 & 0.461 & 0.291 & -0.028 \end{pmatrix}, \\ A_0 &= \begin{pmatrix} A_p & B_pC_{c_1} \\ 0 & A_p + B_pC_c \end{pmatrix} \\ &= \begin{pmatrix} 0.568 & 0.432 & -0.357 & -0.339 \\ 0.432 & 0.568 & -0.142 & -0.134 \\ 0 & 0 & 0.210 & 0.094 \\ 0 & 0 & 0.291 & 0.433 \end{pmatrix}. \end{aligned}$$

As a performance specification for the system $x(t+1) = A_{\sigma_t}x(t)$, we consider the set

$$G_{8,\frac{1}{2}} = \{\sigma : \frac{\|x(t+8)\|}{\|x(t)\|} < \frac{1}{2} \text{ for all } t \in \mathbb{N}, x(t) \in \mathbb{R}^n\}.$$

of all schedules that achieve $(8, 1/2)$-exponential-stability.

In order to get an automaton for this requirement, we compute the set $B = \{\sigma \in \{0,1\}^8 : \|A_{\sigma_8}\cdots A_{\sigma_1}\| \geq 1/2\}$ by enumerating all words of length 8. An automaton that accepts all infinite sequences whose subwords are not in B is depicted in Figure 3. The fact that the structure of this automaton is nontrivial gives some evidence for the value of automata-based specifications.

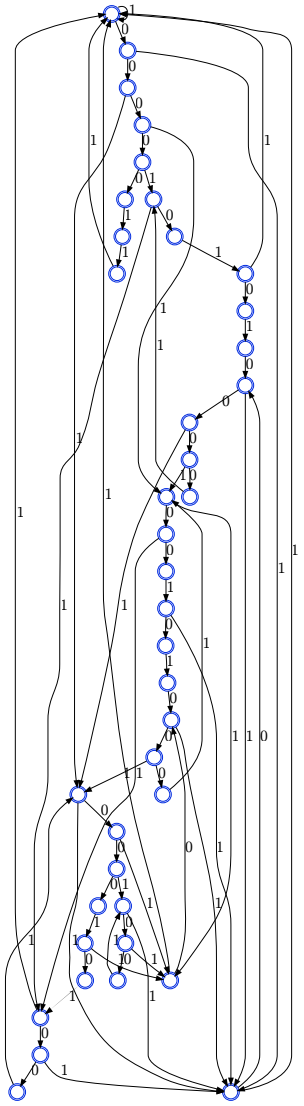

Figure 3. An automaton for $G_{8,\frac{1}{2}}$.

4.2 An online scheduler

To see the power of automata as online schedulers, consider the following experiment: given a number $0 \leq \gamma \leq 1$ (load factor), a random word is generated as follows. The word is generated by a random walk over the graph. If the current state has two outgoing edges - choose 0 with probability γ and 1 with probability $1-\gamma$. If the state has only one outgoing edge, take it.

This experiment simulates a control systems with an online scheduler and the LQG controller scheme described above. The factor $0 \leq \gamma \leq 1$ models an external computation load that the controller has to share computational resources with. The controller has priority over the external load but its load should be proportional to $1-\gamma$. When

$\gamma = 1$, the controller uses the resources only if necessary to guarantee the minimal required performance. When $\gamma = 0$ it takes all available resources. When $0 < \gamma < 1$, it leaves the needed proportion of resources to the external load. In all cases, it is guaranteed that the requirements for the control system are met because the generated sequence is always in $G_{8,1/2}$.

In Figure 4, the output of the plant when simulated with different load factors is displayed. In this simulation, the initial plant state is the equilibrium and the initial estimator state is $(2, 2)^T$. No noise is inserted during the simulation.

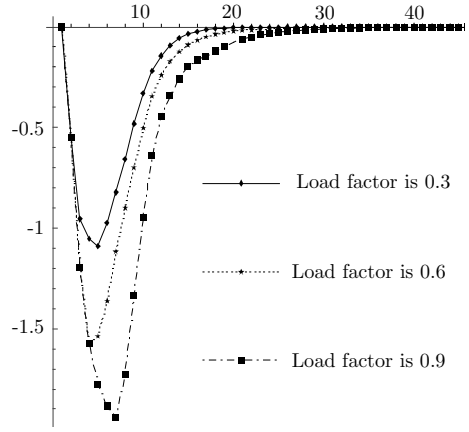

Figure 4. Performance under varying load.

These graphs highlight the advantage of using automata over static scheduling. With static scheduling, one has to plan for the worst case. With automata, it is possible to assure performance in the worst case and adjust the schedule for better performance when resources are available.

An alternative to our approach, provided that the load-factor is confined to a finite number of values, is to design and analyze a schedule for each load value. This solution is only relevant if the load factor does not change too frequently, because transient schedules are not analyzed. Our approach allows frequent mode switches and "infinite" number of modes.

4.3 Schedulability analysis

Often, a single microprocessor is used to implement several control loops. We demonstrate how this can be done with performance guarantees.

As an example, consider a computer that implements three control loops. Assume that these are three independent copies of the plant described in Section 4.1. A block diagram for this system is given is Figure 5.

First, we assume that at most one task can run in each computation slots (as described is Section 2.2). In particular, only one of the control loops can evaluate the output

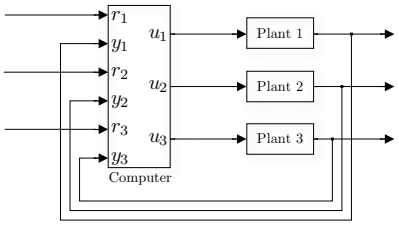

Figure 5. Independent control loops.

of the corresponding plant. The other two loops evolve in simulation mode (the mode in which the current output of the plant is not used, as described in Section 4.1 above).

To compute the composition, we make three copies of the automaton depicted in Figure 3. Then, we replace each 0 in the first automaton with $0, 2, 3$; each 0 in the the second automaton with $0, 1, 3$ and each 0 in the third automaton with $0, 1, 2$. For $i = 1, 2, 3$, we also replace every 1 in the ith automaton with i.

The renamed automata are over the alphabet $\{0, 1, 2, 3\}$. The language of the ith automaton represent the allowed schedules from the point of view of the ith component. To compute the language of scheduled that are allowed by all the subsystems, we take the intersection of these languages.

For this specific case, it turns out that the intersection is empty (the computation takes few seconds on a 2 GHz Intel Core Duo laptop). This means that the system is not schedulable. One option, in such case, is relaxing performance requirements. For example, we can repeat the same procedure for the language $G_{10,1/2}$ instead of $G_{8,1/2}$ (require that any subsequence of length 10 and not 8 is contracting at least by $1/2$). This results with an automaton with 263 states that accepts a non empty language. One can extract a cyclic schedule from this automaton by following a cycle. For example, the schedule $(323121312)^\omega$ is extracted in this way.

Another option, when the system is not schedulable, is a better implementation platform. This corresponds to the analysis method described in Section 2.3. In this case, we replace every i in the ith automaton with the list of subsets of $\{1, 2, 3\}$ that contain i. Zeroes are replaced with the list of subsets that do not contain i.

The intersection of the three set-based automata is an automaton over the alphabet $2^{1,2,3}$ with 8001 states (the computation takes about 30 minutes on a 2 GHz Intel Core Duo laptop). This automaton can be used to choose an implementation platform. For example, we can check if the system can run on a platform that allows any pair of the three tasks to run concurrently by intersecting it with the language $(\{\} + \{1\} + \{2\} + \{1, 2\} + \{1, 3\} + \{3, 2\})^\omega$. This computation takes about about 40 minutes (on the same 2 GHz Intel Core Duo laptop) and yields an automaton with

13121 states and a non-empty language. As another example, we can also check if the system is schedulable on a platform that allows tasks 1 and 2 to run concurrently but task 3 must get an exclusive slot. This amounts to intersecting the requirements automaton with the language $(\{\}+\{1\}+\{2\}+\{1,2\})^\omega$. After another 40 minutes, we get an automaton with a nonempty language, which means that the system is schedulable on such a platform. The resulting automata can be used to extract platform specific schedules. Note that, if the platform is fixed, we can optimize the procedure by only considering the allowed alphabet in the first place.

4.4 Using automata to assure step response properties

In many control applications, a step-response with desired properties is required. We show, as an example, how settling-time can be guaranteed. Consider a system with two modes of operation

$$\begin{aligned} x(t+1) &= A_{\sigma_t}x(t) + B_{\sigma_t}u(t), \\ y(t) &= C_{\sigma_t}x(t) \end{aligned}$$

where

$$\begin{aligned} A_1 &= \begin{pmatrix} 0.37134 & -0.54025 \\ 0.54025 & 0.75658 \end{pmatrix}, \\ B_1 &= \begin{pmatrix} 0.69138 \\ 0.31152 \end{pmatrix}, \\ C_1 &= \begin{pmatrix} 0.24614 & 0.80616 \end{pmatrix} \end{aligned}$$

and

$$\begin{aligned} A_0 &= \begin{pmatrix} -0.37134 & 0.54025 \\ 0.54025 & 0.75658 \end{pmatrix}, \\ B_0 &= \begin{pmatrix} 0.1212 \\ 0.01152 \end{pmatrix}, \\ C_0 &= \begin{pmatrix} -0.5 & 0.3 \end{pmatrix}. \end{aligned}$$

When the loops is closed, the state equation is $x(t+1) = (A_{\sigma_t} - B_{\sigma_t}C_{\sigma_t})x(t) + B_{\sigma_t}r(t)$ where $r(t)$ is the value of the reference signal at time t and $x(t)$ is the state of the system.

The particular requirement that we want to guarantee is

$$\begin{aligned} &G(15, 5, 0.6, 0.86) = \\ &\quad \{\sigma : \text{if } r(0) = \cdots = r(t-1) = 0 \\ &\qquad \text{and } r(t) = \cdots = r(t+15) = 1 \\ &\qquad \text{then } y(t+5), \ldots, y(t+15) \in (0.6, 0.86)\}. \end{aligned}$$

In words: we want the step response to settle in the interval $(0.6, 0.86)$ in 5 steps.

To compute an automaton for this set, we compute the set B of all schedules of length 15 that do not satisfy the above requirement. The computation takes few seconds (on a 2 GHz Intel Core Duo laptop) and give a list of 1174 words (out of $2^{15} = 32768$).

The computation of an automaton whose language are all words that do not contain any substring in B takes 15 minutes (on the same machine) and yields an automaton with 355 states. If we want a periodic schedule of length 4, we can intersect the resulting automaton with the language $\|_{\sigma\in\{0,1\}^4}\sigma^\omega$. This gives the automaton depicted in Figure 6.

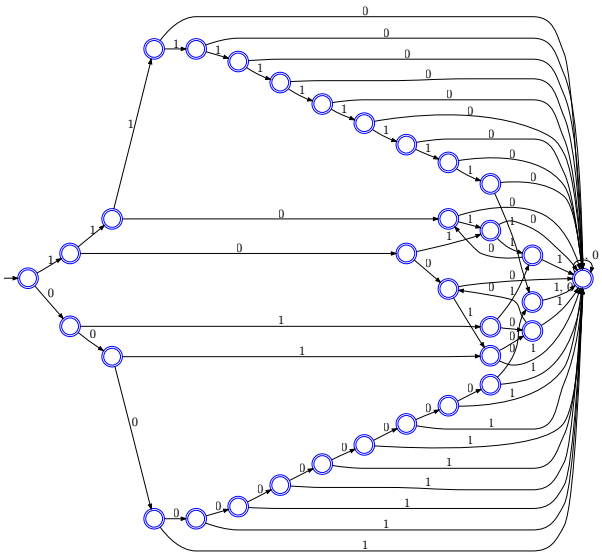

Figure 6. Automaton for periodic schedules that satisfy the settling-time requirement.

5 Conclusions and Future Work

We have illustrated the use of formal languages and Büchi automata over infinite words as an expressive, analyzable, and composable specification framework for resource requirements. In the proposed methodology, the control designer, instead of specifying a fixed period, specifies an ω-regular language L of schedules that are acceptable for control performance. As components are added, their resource specifications are composed using automata theoretic operations, and the system is schedulable as long as the specification language of acceptable schedules in nonempty. We have presented a case study that shows that, for certain performance objectives, automata have the suitable expressiveness, and allow performance varying with the load.

We are designing a prototype implementation for scheduling of control systems. As the number of components grow, we will have to address scalability issues (since checking of emptiness requires constructing the product of

component specifications, followed by the trimming operation). We hope that the symbolic representations developed in the model checking literature will be useful for this purpose. We have recently shown that the performance gap between the model-level semantics of proportional-integral (PI) controllers and their implementation-level semantics can be rigorously quantified if the controller implementation is executed on a predictable time-triggered architecture according to a given periodic schedule [16]. We are exploring if this result can be lifted to ω-regular languages of schedules.

Scanning all possible words, as proposed in the algorithm proposed in Section 3, is not always necessary. Finding better ways to explore the possible evolutions and quantify their performance is a subject for future research. We hope that clever algorithms will allow construction of automata for larger systems and more complicated requirements.

We focus on scheduling a single computational resource, but it is clear that the developed methods can be used for other applications such as scheduling network access and multiple resources. Possible extensions to the framework include distributed and observation-guided scheduling.

Acknowledgments

We thank George Pappas for fruitful discussions related to this paper. This research was partially supported by NSF grants CPA 0541149 and CSR-EHS 0509143.

References

[1] Y. Abdeddaïm and O. Maler. Job-shop scheduling using timed automata. In proc. of 13th Conf. on Computer Aided Verification (CAV), pages 478–492, 2001.

[2] M. Anand, S. Fischmeister and I. Lee. Composition Techniques for Tree Communication Schedules In *19th Euromicro Conference on Real-Time Systems (ECRTS)*, 2007.

[3] S.K. Baruah. A general model for recurring real-time tasks. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 114–122, 1998.

[4] G.C. Buttazo. *Hard real-time computing systems: Predictable scheduling algorithms and applications*. Kluwer Academic Publishers, 1997.

[5] L. de Alfaro and T.A. Henzinger. Interface automata. In *Proc. of the 9th symp. on Foundations of Software Engineering (FSE)*, pages 109–120. ACM Press, 2001.

[6] Z. Deng and J. Liu. Scheduling real-time applications in an open environment. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, pages 308–319, 1997.

[7] C.C Han, K.J. Lin and C.J. Hou. Distance-constrained scheduling and its applications to real-time systems. In *IEEE Trans. Comput.*, 45(7):814826, 1996.

[8] T.A. Henzinger, B. Horowitz, and C.M. Kirsch. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, 2003.

[9] H. Kopetz and G. Bauer. The time triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.

[10] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 2000.

[11] D. Liberzon. *Switching in systems and control*. Birkhäuser, 2003.

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

[13] N.A. Lynch, R. Segala, and F.W. Vaandrager. Hybrid I/O automata. *Information and Computation*, 185(1):105–157, 2003.

[14] A.K. Mok and A.X. Feng. Towards compositionality in real-time resource partitioning based on regularity bounds. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, pages 129–138, 2001.

[15] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems: Specification*. Springer-verlag, 1991.

[16] J. Nilsson, *Real-time control systems with delays*, Ph.D. Thesis, Lund Institute of Technology, 1998.

[16] T. Nghiem, G.J. Pappas, A. Girard, and R. Alur. Time-triggered implementations of dynamic controllers. In *Proceedings of the 6th Annual ACM Conference on Embedded Software (EMSOFT)*, pages 2–11, 2006.

[17] P. Pop, P. Eles, and Z. Peng. Schedulability analysis for systems with data and control dependencies. In *Euromicro Conference on Real-Time Systems*, pages 201–208, 2000.

[18] J. Regehr and J.A. Stankovic. HLS: A framework for composing soft real-time schedulers. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, pages 3–14, 2001.

[19] I. Shin and I. Lee. Compsitional real-time scheduling framework. In *Proceedings of the 25th IEEE Real-Time Systems Symposium*, pages 57–67, 2004.

[20] E.D. Sontag. *Mathematical control theory: Deterministic finite-dimensional systems*, volume 6 of *Texts in Applied Mathematics*. Springer, 1998. Second Edition.

[21] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 133–191. Elsevier Science Publishers, 1990.

[22] M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 1994.

[23] G. Weiss and R. Alur. Automata based interfaces for control and scheduling. In *Proc. 10th Int. Workshop on Hybrid Systems: Computation and Control*, LNCS 4416, pages 601–613. Springer, 2007.

[24] http://www.mathworks.com/products/matlab/