Proceedings of the 2005 Winter Simulation Conference
M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, eds.

AN SDS MODELING APPROACH FOR SIMULATION-BASED CONTROL

Sreeram Ramakrishnan

Engineering Management and Systems Engineering
1870 Miner Circle
University of Missouri–Rolla
Rolla, MO 65409, U.S.A.

Mayur Thakur

Department of Computer Science
1870 Miner Circle
University of Missouri–Rolla
Rolla, MO 65409, U.S.A.

ABSTRACT

We initiate a study of mathematical models for specifying (discrete) simulation-based control systems. It is desirable to specify simulation-based control systems using a model that is intuitive, succinct, expressive, and whose state space properties are relatively easy computationally. We compare automata-based models for specifying control systems and find that all systems that are currently used (such as finite state machines, communicating hierarchical finite state machines (FSM), communicating finite state machines, and Turing machines) lack at least one of the abovementioned features. We propose using sequential dynamical systems (SDS)—a formalism for representing discrete simulations—to specify simulation-based control systems. We show how to adapt the standard SDS model to specify cell-level controllers for a generic cell. For reasonable flexible manufacturing cells, the SDS-based specification has size polynomial in the size of the cell, while in the worst case the FSM-based specification has size exponential in the size of the cell.

1 INTRODUCTION

Simulation-based control is an approach for building controllers in which the controller makes decisions based both on the current state of the system as well as the result of "likely future scenarios." These likely future scenarios are determined by carrying out simulations. (Thus, the name "simulation-based control.") It is not difficult to imagine scenarios in which simulation-based control leads to a more efficient system, and thus to higher industrial productivity. Even though simulation-based control is an attractive paradigm conceptually, there are technical issues that need to be addressed if we want simulation-based control to be more efficient than traditional control. The chief technical issue is efficiently computing the "likely future scenarios." That is, it is desirable that simulations that are carried out are themselves efficient. This motivates the following question:

> How can distributed control systems be formally modeled so that (a) the representation is succinct, (b) the representation is expressive (that is, it can be used to represent a large class of distributed control systems) and, (c) the model state space is somewhat easy to (computationally) analyze?

In this paper we take first steps toward answering this question. We argue that in the wide spectrum of automata-based models available to us between the two extremes—finite-state machines and Turing machines (or programs)—models that have been used to date lack at least one of the properties mentioned above. For example, finite state machines are not succinct and almost any interesting question regarding programs is undecidable. Thus, the issue of *finding* a suitable formal model is itself interesting.

We consider three models—communicating finite state machines (CFSM) (Brand and Zafiropulo 1983), communicating hierarchical finite state machines (CHM) (Alur et al. 1999), and sequential dynamical systems (SDS) (Barrett et al. 2000). A CFSM is a collection of finite state machines (representing processes) with each communicating pair of processes connected by a duplex first-in-first-out channel. A CHM is a generalization of finite state machines in which a state is either a simple state or is itself a CHM. An SDS consists of a graph with each node having a local transition function whose inputs are the states of neighbors of the node (and its own state). At each time step, the states are updated sequentially in prescribed order.

These models have been used for modeling protocols, theoretically analyzing simulations, and formalizing design specification. We argue, however, that though both CFSM and CHM are succinct, neither meets all the criteria mentioned above. CFSM can simulate Turing machines, and so its state space is impossible to analyze computationally. The expressiveness of of CHMs is equivalent to that of FSMs. However, reachability in CHMs is intractable.

We propose a modification of the standard SDS model that is intuitive, succinct, and expressive. We call this model the *Input/Output SDS* model or IO-SDS for short. We conjecture that for some restricted cases state space reachability in IO-SDS is tractable (even though for other cases state space reachability in SDS is intractable.) To show the applicability of our models, we show how to specify a flexible manufacturing cell (FMC) using SDS.

Note that all the automata-based models are relatively simple and it can be argued that we are trying to force-fit distributed control systems into these automata-based models that were not designed with distributed control systems in mind. One might wonder whether this force-fitting is warranted. Instead of this approach, would finding a formal model that is representative of the distributed control systems not be more reasonable? The reason we want to consider automata-based models is two-fold. First, the automata-based models are well understood and if we use automata-based models for distributed control systems, we can use known (theoretical) results on these automata-based models to, for example, increase the efficacy of simulation-based control. Second, the simplicity (and thus, the universality) of the automata-based model is an advantage because a complex model is hard to analyze mathematically. Thus, a model that is formal, yet mimics distributed control systems would likely be much harder to analyze than the automata-based models.

2 FORMAL MODELS FOR SIMULATION-BASED CONTROL

Traditionally, simulation models have been used extensively for the purpose of long-term design and analysis of flexible manufacturing systems. The successful use of simulation as a design tool has encouraged researchers to investigate into possible ways and methods of using these models for short-term planning, scheduling, and control (Stecke, 1988). Although the basic manufacturing modeling requirements are the same in both design and control applications, some basic differences in their requirements exist (Davis et al. 1991). The area of real-time control of manufacturing cells using simulation models has been the subject of a number of research projects, such as those described in Manivannan and Banks (1991), McConnell and Medeiros (1992), Smith et al. (1994), Drake et al. (1995), and Son et al. (1999). Smith et al. (1994) examined the application of discrete event simulation to shop floor control of a flexible manufacturing system. Here, simulation is used as a task generator to control the physical equipment on the shop floor. Wu and Wysk (1988) created a "multi-pass framework" for simulation-based control by combining a learning system with simulation. Wu and Wysk (1989) later presented a multi-pass scheduling algorithm using a mechanism controller and a flexible simulator. Multi-pass scheduling algorithms are defined as the scheduling algorithms that deal with the scheduling problem of selecting the best dispatching rule among rules in an alternative space. Son et al. (1999) documented the implementation of a multi-pass simulation-based, real-time scheduling and shop floor control system (using a single model).

In general, four types of control architectures have been developed and researched (Dilts et al. 1991): (i) centralized control; (ii) hierarchical control, (iii) heterarchical/distributed control, and (iv) hybrid control. For each of the above architectures, implementation specifics for simulation-based control need to be developed associatively. The implementation of simulation-based control architecture for shop floor control has been partitioned into planning, scheduling, and execution functions (Jones and Saleh 1990). For each of the three functions, wide research has been conducted, and the complexities of each individual function are also well known.

Smith and Joshi (1995) developed a formal model of the execution portion of shop floor control. It is to be noted that the execution activities were completely separated from the decision-making activities, enabling different decision-making strategies to be implemented depending on the situations. Execution has been defined as a function governing what (tasks), where (equipment), and how (methods, e.g. messaging) in shop floors. Similarly, Smith et al. (1994) define execution as verifying the physical preconditions and communicating with the subordinate systems to facilitate the tasks. Smith et al. (1994) state that since the execution module causes physical action, there must be a mechanism for physically verifying certain preconditions to prevent catastrophic errors in the system. Techniques that have been used to implement the execution model include Petri-nets (Kasturia et al. 1988), push-down automata (Mettala 1989), finite state automata (Smith and Joshi 1995), object oriented methods, graphical representation (Cho and Wysk 1995) and programmable logic control. Common information embedded in these techniques is composed of two sets: specifications for each state (condition), and a set of variables and associated actions that change the state.

In fact, models that mimic control systems have been proposed, for example, by Joshi et al. (1995). However, we reemphasize that our approach is not to propose a new model that mimics control systems, but to use existing models (with as little modification as possible) so that results on these existing models can be carried over to our work.

3 RESEARCH OBJECTIVES

The main objective of this research is to develop SDS formal models that will allow further investigation into the use of communicating hierarchical state machines in formally

modeling distributed and hierarchical control systems. The focal domain of this paper is discrete-part manufacturing, a domain where simulation-based control has been successfully implemented by previous researchers.

The paper presents an *SDS formal modeling approach* for developing control architectures for discrete-part manufacturing systems. The modeling methodology is discussed with an example manufacturing cell. In addition to the modeling constructs, the paper discusses issues related to reachability analysis, state space generation, and complexity analysis, specifically *comparing it with finite state automata models.*

As discussed earlier, finite state representations have been extensively used to formally model discrete control systems. In addition, such formalisms have been the basis for generating simulation-based control architectures. More recently, formalisms such as finite state automata were also the basis for developing methodologies to automatically generate DES models that can be used for real time control and for traditional "fast" analysis.

A modeling method that focuses on generating simulation constructs for real time control need to focus on three specific sub-models: object model, interaction model, and logic. This paper discusses the *control logic component* that need to be incorporated in methods to generate DES models for control. A detailed discussion on the sub-models using eXtensible Markup Language (XML) is available in Srinon and Ramakrishnan (2005).

The model discussed in this paper will be extended to include distributed systems and/or hierarchy. Such a method will be based on "communicating hierarchical state machines". The discussed model and the extensions will enable generating DES models for distributed/hierarchical control.

4 MODELING FOR SIMULATION-BASED CONTROL

A traditional controller (for, say, a flexible manufacturing cell) can be thought of as a decision-making algorithm that makes real-time decisions (such as which route through the cell a given part should take at a given moment) taking into account only the current state of the system. However, a decision that is optimal for the current state might not be optimal in the "long run." For example, if certain events happen in the future the decision made now might turn out to be a bad decision. A simulation-based controller can be thought of as a decision-making algorithm that makes real-time decision based on the current state of the system and the state (or states) the system is likely going to be in the future. Thus, roughly speaking, simulation-based controllers make decisions by looking ahead in the future. Given that the system is in state s_0 at time $t = 0$, the number of possible states that the system could be in at time $t = \alpha$ is, in general, exponential in α. This is because at each time step some events can take place. In general, these events take place independently of each other and so the number of possible states that the system could be in at time $t = \alpha$ is exponential in α. This means that a simulation-based controller cannot hope to both provide real-time control and look too far ahead in the future because, roughly speaking, there are too many possible scenarios to consider. Note that the "look ahead" to time $t = \alpha$ corresponds to finding which states are reachable from state s_0. If state space reachability, which is provably hard for many interesting systems, can be solved efficiently, then the look ahead in simulation-based controllers can be speeded leading to better simulation-based controllers. The hardness that we refer to here can be given a technical meaning depending on the system. For example, for Turing machines state space reachability is undecidable. For Sequential Dynamical Systems (SDS), which are described in Section 5, even if the functions used in the nodes of the SDS are all simple threshold functions, the state space reachability is PSPACE-complete (Barrett et al. 2003).

Our approach in this work is based on the following observations: First, state space reachability is a hard problem in general. However, in practice many systems might be simple enough that state-space reachability can be solved relatively easily. Second, in real-world systems there could be initial states such that it is easy to compute which states are reachable from these initial states. Third, optimization versions of state space reachability problems might have fast approximation algorithms which might suffice in practice.

It appears then that complexity of state space reachability for a particular model should be a key factor in deciding whether that model is suitable for simulation-based control. But should state space reachability be the only factor? Or do we need to take into account other factors as well? It is not difficult to see that other factors need to be taken into account as well, most crucially, perhaps, the following two:

1. Is the model succinct (that is, the representation of any system in the model is relatively small)?
2. Is the model expressive (that is, the model can be used to represent a wide variety of systems)?

To see why succinctness is relevant, consider two models A and B such that state space reachability for A is NP-hard, while state space reachability for B is in P. Can we conclude that B is a better model than A? The answer is "no." To see this, consider the following scenario: For any system X, if the representation of X as a model of type A has size n, while the representation of X as a model of type B has size 2^{2^n}. Now the time that the state space reachability algorithm takes is $\mathcal{O}(2^{p(n)})$ when the system is represented as a model of type A and is $\Omega(2^{2^n})$ when

the system is represented as a model of type B. Thus, A is in fact a better model than B.

The reason expressiveness is important is that we want the techniques we develop to hold for a large class of systems. What sort of control systems should we be able to handle? In this paper, we say that the "universe" of systems that we want to model is the (cell-level) control for FMCs. (In Section 4.1, we precisely define our universe.)

One might ask why we need a new model of controllers when several such models have already been proposed in the literature? The answer is rather simple: *The models for controllers proposed in the literature have not been designed to make analyzing state space reachability easy.* These models have been designed for different puroposes. For example, Joshi et al. (1995) proposed a model (for FMC control) that makes automatic generation of control software possible. The DEVS formalism of Zeigler et al. (2000) (see also Concepcion and Zeigler 1988) for discrete event systems allows specification in a hierarchical manner. Radiya and Sargent (1987) take a programming language approach and define a formalism for discrete event specification that includes a rigorous specification of the semantics of the model in addition to syntactic constructs of the specification language used to specify the simulation. Radiya and Sargent (1994) present a modal logic based formalism to specify and implement discrete event simulations.

The SDS-based model that we descibe below is designed keeping in mind the ultimate application of this model. In particular, we want to be able to solve the reachability problem quickly in some restricted cases. Thus, identifying which cases are easy and which are hard is important. In this paper, we describe a formal model that is succinct and based on SDS. Considerable work has been done in separating the easy and hard cases of reachability problems in SDS. We want to adapt these known results in our case with the goal of designing better simulation-based controllers.

In Section 4.2, we look at automata-based systems—finite state machines, communicating finite state machines, and communicating hierarchical machines—that have been proposed as formal models for real-time controllers, communcations protocols, and flow of control in software control. We will argue that each of these systems lacks some feature that is desirable in a model for an effective simulation-based controller. Before we discuss these shortcomings, we introduce in Section 4.1 an example system that we will use to discuss these shortcomings.

4.1 Example FMC

A flexible manufacturing cell (FMC) is a manufacturing system that can produce multiple types of parts, where a routing sequence (or a set of sequences) is associated with each part. A routing sequence through a system is just a sequence of subparts of the system.

For the purposes of this paper, the "universe" is the set of FMCs, formally defined as:

Definition 1 *For integers $n_1 > 0$, $n_2 > 0$, $n_3 > 0$, $k > 0$, and $\ell > 1$, a (n_1, n_2, n_3, k, ℓ)-*FMC *is a flexible manufacturing cell consisting of the following parts:*

1. *n_1 ports,*
2. *n_2 material processors, and*
3. *n_3 material handlers (or robots).*

The system handles k types of parts. Each part has exactly one prescribed route through the cell of length ℓ. The route *of a part type is a sequence of ports and material processors that begins and ends with a some port.*

When the parameters of an (n_1, n_2, n_3, k, ℓ)-FMC are not important or they are clear from the context, we will denote simply as "FMC."

It should be noted that the classification of equipment types in the FMCs has been adopted from resource models discussed in Steele et al. (2001). The different components of a shop floor modeled as a resource can be represented using equipment (E), instruction sets (I), and a connectivity graph (CG), which are defined as follows: $R = E \cup I \cup CG \cup \ldots$. In the above definition, the "..." denotes that the scope of R varies depending on the nature of the application. For example, in the case of a shop floor, the above definition can be expanded to include tools (T), fixtures (F), locations (L), ports (P), and facilitators (FA) as $R = E \cup T \cup F \cup I \cup CG \cup L \cup P \cup FA \cup \ldots$. These resource models are not dependent on implementation and serve as a starting point for designing control systems. DES models provide one method of utilizing these models for control system design. The methodology of generating a single DES from resource models was demonstrated in Son (2000).

In Section 5.3, we describe how to model a controller for FMC using an IO-SDS. We explain the modeling using an example $(1, 2, 1, 2, 4)$-FMC depicted in Figure 1. The example FMC consists of the following equipments: a port (equipment 0), a robot (equipment 1), and two material processors (equipments 2 and 3). There are two types of parts: type 0 and type 1. The prescribed route for parts of type 0 is (0, 2, 3, 0). (That is, parts of type 0 arrive at the port, then they are processed by the first material processor, followed by the second material handler, and finally they are kept back at the port.) The prescribed route for parts of type 1 is (0, 3, 2, 0).

4.2 Modeling Systems Using Automata

Let us briefly look at the key automata-based models that could potentially be used in simulation-based control. The models that we consider are: finite state machines (FSM), communicating finite state machines (CFSM), communicating hierarchical state machines (CHM), and Turing machines. Below we informally describe each of these models

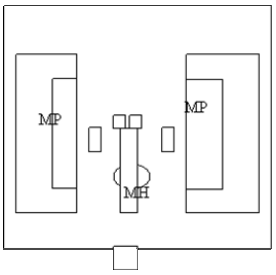

Figure 1: A (1, 2, 1, 2, 4)-FMC

and discuss the advantages/disadvantages of using these models as controllers in simulation-based control.

The use of finite state machines (FSMs) as controllers is not new. In fact, several formalisms (such as DEVS (Zeigler et al. 2000)) that have been proposed for specification of discrete event simulation use FSMs. The reason FSM is an attractive model is (a) it is simple (b) it captures the essence of finite-state systems such as a discrete event simulation and (c) it can be implemented in hardware. However, of the three desirable properties (succinctness, epressiveness, and easiness of state space reachability), FSM lacks one, namely, it is not succinct. In particular, the number of states in an FSM controller for an FMC can be exponential in the number of ports/equipments/robots. It is easy to see this: In general, an FSM controller for an (n_1, n_2, n_3, k, ℓ)-FMC must carry at least the following information in each state:

1. Which of the ports/equipments/robots contain parts? This requires $n_1 + n_2 + n_3$ bits of information.
2. For each part in the cell, what is its part type? This requires $(n_1 + n_2 + n_3) \log k$ bits of information.
3. For each part in the cell, how many equipments has the part already been through? This requires $(n_1 + n_2 + n_3) \log \ell$ bits of information.

Since, in general each configuration of the FMC is reachable, there are a total of $2^{(n_1+n_2+n_3)(1+\log k+\log \ell)}$ states in an FSM controller for an (n_1, n_2, n_3, k, ℓ)-FMC.

A communicating finite state machine (CFSM) (Brand and Zafiropulo 1983) is a model used to describe distributed protocols. It has multiple finite state machines $M_1, M_2, \ldots, M_k$ communicating with each other via channels. The state of a machine M_i can change due to one of two reasons: it receives a message from another machine or it transmits a message to another machine. Brand and Zafiropulo argue that it is natural to consider the channels to be of infinite capacity because even though in practice the channels have finite capacities, the bounds on the capacities are way too large to be practical. If we consider infinite capacity channels, then a CFSM can essentially simulate a Turing machine and thus reachability in CFSM is undecidable. If we

Informally, in a communicating hierarchical state machine (CHM) (Alur et al. 1999), a (super)state can itself be a machine. Thus the machine is hierarchical. In additional, there may be several sub-machines within a CHM that run in parallel. These machines communicate (synchronize) using symbols that are common alphabet symbols. The expressiveness of CHMs is equivalent to that of FSMs. CHMs offer succinctness in some cases. However, reachability in CHMs is EXPSPACE-complete (Alur et al. 1999), and thus intractable.

A Turing machine provides great expressiveness: There are controllers that can be represented as a Turing machine but not as finite-state machines. However, the reachability problem for Turing machines is undecidable. Thus, Turing machines are unsuitable for our purposes.

5 SDS-BASED MODEL FOR SIMULATION-BASED CONTROL

As we have seen above, automata-based models that have been used to specify and analyze industrial control systems lack either succinctness or expressiveness. In this section, we show how we can alleviate these problems by using a modification of sequential dynamical systems to model industrial control systems. We first introduce the standard SDS model and then introduce a modification (IO-SDS) of the SDS model that we use to model control systems. We show how the example cell controller discussed in Section 5.3 can be modeled using an IO-SDS.

5.1 Standard SDS Model

Sequential Dynamical Systems (SDS) is a theoretical framework for analyzing discrete simulations. The standard SDS model consists of the following three components: an underlying graph $G = (V, E)$, a set of local update functions $\mathcal{F}$, and an update order π. (We formalize the SDS model in Definition 2.) Each node in V denotes an "element" in the simulation. For our purposes, it is helpful to think of each node as representing a "variable" of the system we are simulating. The edges in E denote the dependencies between variables. Thus, an edge between variables x and y denotes that "x and y are interdependent." Each node (variable) x has an update rule (function) associated with it. The function f associated with x captures how x depends on other variables. The inputs to f are the current values of x and the values of its neighbors (in G). The set of all these functions forms $\mathcal{F}$. π is a permutation on the nodes and it denotes that order in which the nodes are updated at any time step.

Definition 2 *(Barrett et al. 2000) A* sequential dynamical system *(abbreviated as SDS) over domain $\mathcal{D}$ is a tuple $\mathcal{S} = (G, \mathcal{F}, \pi)$ such that the following hold:*

1. *$G = (V, E)$ is an undirected graph with n nodes labeled $1, 2, \ldots, n$. G is called the* underlying graph *of $\mathcal{S}$.*
2. *$\mathcal{F} = (f_1, f_2, \ldots, f_n)$ is an ordered set of boolean functions such that, for each $i = 1, 2, \ldots, n$, the following holds: If the degree of node i is k, then f_i is a function of type $\mathcal{D}^{k+1} \rightarrow \mathcal{D}$. For each $i = 1, 2, \ldots, n$, f_i is the called the* local transition function *for node i.*
3. *π is a permutation on $\{1, 2, \ldots, n\}$ known as the* update order *of $\mathcal{S}$.*

Let $\mathcal{S} = (G, \mathcal{F}, \pi)$ be a SDS over domain $\mathcal{D}$. Let $G = (\{1, 2 \ldots, n\}, E)$. A *configuration* of S is an element of $\mathcal{D}^n$. Let $C = (s_1, s_2, \ldots, s_n)$ be a configuration. Let $i \in \{1, 2, \ldots, n\}$. We say that node i is in state s_i in configuration C. We now describe how configurations of SDS evolve over time. Starting in an initial configuration (set of values to assigned to the nodes of an SDS), an SDS $S = (G, \mathcal{F}, \pi)$ *evolves* as follows. Each time step consists of n mini-steps. In the ith mini-step, the state of node i is updated to $f_i(\cdot, \cdot, \ldots, \cdot)$, where the inputs to f_i are the the current state of node i and the current states of neighbors of i. (Note that if a neighbor of i, say j, was updated in a mini-step before i, then the input to f_i would be the updated state of j.)

More formally, let C be the configuration of $\mathcal{S}$ at time step t. Then the configuration of $\mathcal{S}$ at time step $t + 1$ is denoted by $\text{next}_{\mathcal{S}}(C)$, and it is defined as follows. We first define a series of configurations $C_0, C_1, C_2, \ldots, C_n$. Let $C_0 = C$. C_k represents the configuration after mini-step k. C_k is obtained from C_{k-1} in the following manner. For each $i \in \{1, 2, \ldots, n\}$, let s_i be the state of node i in configuration C_{k-1}. Let $\ell = \pi(k)$. Let X be the neighbors of node ℓ in G. Let $x_1, x_2, \ldots, x_m$ be the elements of $X \cup \{\ell\}$ in increasing order. Let $s' = f_\ell(s_{x_1}, s_{x_2}, \ldots, s_{x_m})$. Then, $C_k = (s_1, s_2, \ldots, s_{\ell-1}, s', s_{\ell+1}, s_{\ell+1}, \ldots, s_n)$. Now, $\text{next}_{\mathcal{S}}(C) = C_n$.

The sequentiality of updates is a crucial difference between SDS and models such as neural networks, cellular automata, and communicating finite state machines, all of which use synchronous (i.e., parallel) updates. Note that for the same underlying graph and the same local transition functions, different update orders may produce different dynamic behavior. onfiguration at the next step depends on the current configuration, the structure of the underlying graph, the local transition functions, and the update order.

Example 1 *Consider an SDS $\mathcal{S}_1$ defined over the boolean domain as follows. $\mathcal{S}_1 = (G, \mathcal{F}, \pi)$, where G is the 4-node graph shown in Figure 2(a), $F =$* [NOR, AND, AND, NOR]*, and $\pi = [1, 2, 3, 4]$. Figure 2(b) shows the configuration, $\mathcal{C}_0$, of $\mathcal{S}_1$, say, at step t. Note that in $\mathcal{C}_0$, nodes 1, 3, and 4 are in state 0, while node 2 is in state 1. Figures 2(c) and 2(d) show the configurations ($\mathcal{C}_1$ and $\mathcal{C}_2$) of $\mathcal{S}_1$ at steps $t + 1$ and $t + 2$, respectively.*

SDS has been used to design simulations for a plethora of real-world systems. Examples of such systems include TRANSIMS (urban traffic simulation) (Barrett et al. 2001), EpiSims (simulation of the spread of epidemics) (Eubank et al. 2004), AdHopNET (simulation of packet-switched communication systems) (Barrett et al. 2004), and Marketecture (simulation of deregulated electrical power markets) (Atkins et al. 2004).

Reachability in SDS has been studied. In particular, Barrett et al. (2003) show that the reachability problem in SDS that use symmetric threshold functions as their transition functions is polynomial-time computable (and thus, easy). On the other hand, SDS for asymmetric threshold functions is PSPACE-hard.

5.2 Input/Output SDS

SDS model dynamical systems in which the there are no external inputs and there are no explicit outputs. In order for us to be able to use SDS to specify control systems (such as a cell-level controller), we need to adapt the standard SDS model so that inputs and outputs are explicitly modeled. We define *Input/Output Sequential Dynamical Systems* (abbreviated as IO-SDS) as a natural extension to the standard SDS model. The underlying graph in an IO-SDS contains two types of special nodes: *input nodes* and *output nodes*. The input nodes are *not* updated during the update process, though their values can be set externally. The values of output nodes can be read externally.

Definition 3 *An* input/output sequential dynamical system *(abbreviated as IO-SDS) over domain $\mathcal{D}$ is a tuple $\mathcal{S} = (G = (V \cup I \cup O, E), \mathcal{F}, \pi)$ such that the following hold:*

1. *V (*internal nodes*), I (*input nodes*), and O (*output nodes*) are mutually disjoint sets such that $||V|| = n_V$, $||I|| = n_I$, and $||O|| = n_O$.*
2. *$G = (V \cup I \cup O, E)$ is an undirected graph with $n_I + n_V + n_O$ nodes. Let $n = n_V + n_O$. The set of nodes in $V \cup O$ are labeled $1, 2, \ldots, n$. G is called the* underlying graph *of $\mathcal{S}$.*
3. *E contains no edge from $\{(x, y) \mid x, y \in I\} \cup \{(x, y) \mid x, y \in O\}$.*
4. *$\mathcal{F} = (f_1, f_2, \ldots, f_n)$ is an ordered set of boolean functions such that, for each $i = 1, 2, \ldots, n$, the following holds: If the degree of node i is k, then f_i is a function of type $\mathcal{D}^{k+1} \rightarrow \mathcal{D}$. For each $i = 1, 2, \ldots, n$, f_i is the called the* local transition function *for node i.*

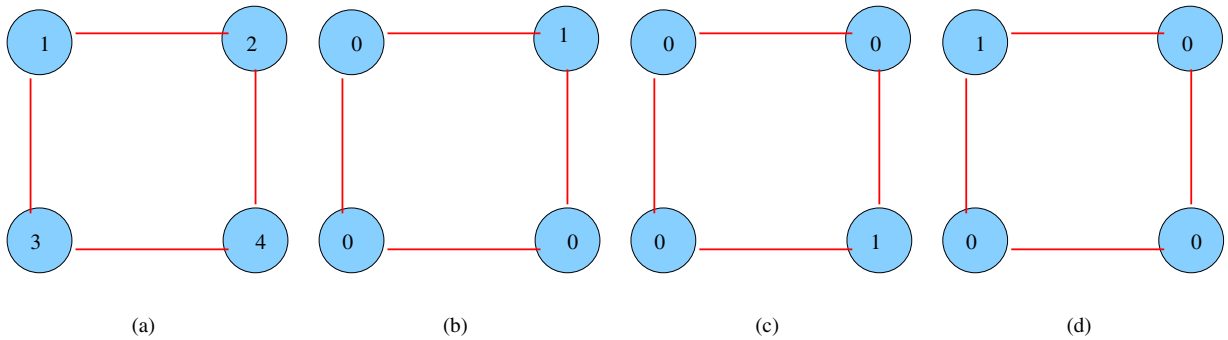

Figure 2: (a) The Underlying Graph for SDS $\mathcal{S}_1$ (b) Configuration, $\mathcal{C}_0$, of $\mathcal{S}_1$ at Step t (c) Configuration, $\mathcal{C}_1$, of $\mathcal{S}_1$ at Step $t+1$ (d) Configuration, $\mathcal{C}_2$, of $\mathcal{S}_1$ at Step $t+2$

5. π *is a permutation on* $\{1, 2, \ldots, n\}$ *known as the* update order *of* $\mathcal{S}$.

Each step of an IO-SDS works exactly like in an SDS except only the internal and output nodes are updated (according to the update order); the input nodes are not updated at all. The values of the input nodes are set externally.

5.3 Using IO-SDS for Control

We now show how a controller can be specified using an IO-SDS. First, we specify an IO-SDS for the example shown in Figure 1. Later, we will show that a large class of controllers can be represented as IO-SDS.

We now describe the IO-SDS $\mathcal{S} = (G = (I \cup V \cup O, E), \mathcal{F}, \pi)$ by describing sets I, V, O, E, $\mathcal{F}$, and π. In the labels of nodes, 0 stands for the *port*, 1 for the *robot*, and 2 and 3 stand for the two *material processors*. Thus, the node labeled robotDest2 in the IO-SDS denotes whether the equipment 2 (that is, the first material processor) is the destination for a part movement by the robot.

Input nodes I: The input nodes of the IO-SDS correspond to the input signals from the 4 equipment-level controllers.

- Equipment 0 (port): The inputs from the port are:
  - $empty_0$: denotes whether the port is empty ($empty_0 = 1$) or nonempty ($empty_0 = 0$).
  - $receptive_0$: denotes whether the port is receptive (objects can be put on the port) or not.
  - $partType_0$: denotes the type of part (type 0 or type 1) on the port. We assume, for this example, that there are only two types of parts. However, extending the model to handle the case when there are more than 2 part types is fairly straightforward. Note that the signal $partType_0$ is "valid" only if the port is nonempty ($empty_0 = 0$).
- Equipment 1 (robot): The inputs from the robot are:
  - $idle_1$: denotes whether the robot is idle.
  - $inProg_1$: denotes whether the robot is in motion.
  - $done_1$: denotes whether the robot has just finished moving an object from one location to another.
- Equipments 2 and 3 (material processors): The inputs from the material handlers are:
  - $idle_2$, $idle_3$: denotes whether the material processor is idle.
  - $inProg_2$, $inProg_3$: denotes whether the material processor is processing a part.
  - $done_2$, $done_3$: denotes whether the material processor has finished processing a part.

Internal nodes V:

- For each $i = 0, 2, 3$, $partType_i$ denotes the type of part in equipment i. $partType_i$ depends on $done_1$, $robDest_i$, $robSrc_0$, $robSrc_2$, $robSrc_3$, and for each $j \in \{0, 2, 3\}$, $partType_j$. (These variables are defined below.) If $done_1 = 1$ and $robDest_i = 1$, then $partType_i = partType_j$, where j is the smallest integer in $\{0, 2, 3\}$ such that $robSrc_j = 1$. If $done_1 = 0$ or if $robDest_i = 0$, then $partType_i$ remains unchanged.
- For each $i = 0, 2, 3$, hop_i the number of times the part in equipment i has been processed by some equipment in the current cell. Thus, when a part of type 0 (route: 0, 2, 3, 0) is on equipment 3, $hop_3 = 2$. hop_i depends on $done_1$, $robDest_i$, $robSrc_0$, $robSrc_2$, $robSrc_3$, hop_0, hop_2, and hop_3. If $done_1 = 1$ and $robDest_i = 1$, then $hop_i = hop_j + 1$, where j is the smallest integer in $\{0, 2, 3\}$ such that $robSrc_j = 1$. Otherwise (that is, if $done_1 = 0$ or $robDest_i \neq 1$), hop_i remains unchanged.

- For each $i = 0, 2, 3$, $next_i$ is the equipment number that the part currently in equipment i needs to be processed in next. Clearly, $next_i$ depends on $partType_i$ and hop_i. $next_i = next(partType_i, hop_i)$, where $next$ is the function the maps a part type t and an integer r to the rth equipment in the route that parts of type t need to take in the cell.
- For each $i = 0, 2, 3$, $nextEmpty_i$ denotes the status (empty/nonempty) of the equipment that the part currently in equipment i needs to be processed in next. Clearly, $nextEmpty_i$ depends on $next_i$ and input nodes $empty_0$, $receptive_0$, $idle_2$, and $idle_3$. If $next_i = 0$, $nextEmpty_i = empty_0 \wedge receptive_0$. If $next_i = 2$, $nextEmpty_i = idle_2$. If $next_i = 3$, $nextEmpty_i = idle_3$.

Output nodes O:

- For each $i = 0, 2, 3$, $robSrc_i$ denotes whether the robot is to be instructed to move the part currently in equipment i. $robSrc_i$ depends on $nextEmpty_i$, $done_i$, and $robSrc_j$, for each $j \in \{0, 2, 3\}$ such that $j < i$. The $robSrc_j$'s are required because if there are multiple parts that are ready to be moved, we need to prioritize them. (We are assuming that parts in 0 get priority over parts in 2, which in turn get priority over parts in 3.) $robSrc_i = 1$ if and only if $done_i = 1$ and $nextEmpty_i = 1$ and, for each $j \in \{0, 2, 3\}$ such that $j < i$, $robSrc_j = 0$.
- For each $i = 0, 2, 3$, $robDest_i$ denotes whether the robot is to be instructed to move some part to equipment i. $robDest_i$ depends on $robSrc_0$, $next_0$, $robSrc_2$, $next_2$, $robSrc_3$, and $next_3$. If $robSrc_0 = 1$ and $next_0 = i$, then $robDest_i = 1$. If $robSrc_2 = 1$ and $next_2 = i$, then $robDest_i = 1$. If $robSrc_3 = 1$ and $next_3 = i$, then $robDest_i = 1$.

Note that in addition to the sets I, V, and O, we have also defined above the set E of edges between these nodes. In particular, when we say above that "x depends on y" we mean that there is an edge (x, y) in E.

The update order π is:

1. for $i = 0, 2, 3$, $partType_i$,
2. for $i = 0, 2, 3$, hop_i,
3. for $i = 0, 2, 3$, $next_i$,
4. for $i = 0, 2, 3$, $nextEmpty_i$,
5. for $i = 0, 2, 3$, $robSrc_i$, and
6. for $i = 0, 2, 3$, $robDest_i$.

What is the size of our IO-SDS-based specification of FMC controllers? Note that the size of the underlying graph and update order is polynomial $\mathcal{O}((n_1+n_2+n_3)(1+\log k+\log \ell))$. However, we also need to include the size of $\mathcal{F}$, the internal functions. The representation of functions in $\mathcal{F}$ can indeed be exponential in the number of its input. To see this, consider a function with n binary inputs such that the concatenation of the output bits is a string of high Kolmogorov complexity (see Li and Vitányi 1997). Thus, any description of such a function will be exponential in n. Thus, there are controllers for which the size of any IO-SDS that represents the controller is $\Omega(n_1+n_2+n_3)$. However, note that controllers in the real world tend to be simple and lightweight. Thus, it is reasonable to assume that the local transition functions in $\mathcal{F}$ have descriptions that are polynomial in the size of the number of inputs. We call such functions *reasonable functions*. An FMC controller which can be represented as an IO-SDS with reasonable transition functions a *reasonable FMC controller*. Thus, we have that *all* reasonable FMC controllers have IO-SDS representation that is polynomial in n_1, n_2, n_3, k, and ℓ. In contrast, there are reasonable FMC controllers whose FSM representation is exponential in $n_1 + n_2 + n_3$.

6 CONCLUSION: TOWARD HIERARCHY IN SIMULATION-BASED CONTROL

In this paper, we presented preliminary work toward formalism that will facilitate efficient simulation-based control. In particular, we introduced Input-Output Sequential Dynamical Systems as one such formalism. We note, however, that IO-SDS are far from being perfect representations. First, we do not know the reachability properties of IO-SDS. Characterizing the easy and hard cases for reachability in IO-SDS is both interesting theoretically and from a practical standpoint. Second, in the current definition of IO-SDS one needs to specify all the components of separately. With the present definition, we cannot exploit the regularity of structure of FMCs that occur in the real world. In particular, we would want to incorporate hierarchy in the definition of IO-SDS. The definition and specification is itself an interesting area of research not the least because the ultimate goal in this line of research is efficient simulations. In particular, we would want the reachability problem in our hierarchical IO-SDS to be somewhat tractable (at least in meaningful restricted cases). Given that the general reachability problem is PSPACE-hard for SDS, we are interested in approximation algorithms for appropriate optimization versions of the reachability problem. In this context, the work of Marathe et al. (1998) on approximation algorithms for hierarchically specified graph problems seems interesting.

REFERENCES

Alur, R., S. Kannan, and M. Yannakakis. 1999. Communicating hierarchical state machines. In *Automata, Languages and Programming 26th ICALP'99*, ed. J. Wiedermann,

P. van Emde Boas, and M. Nielsen, Volume 1644 of *LNCS*, 169–178.

Atkins, K., C. Barrett, C. Homan, A. Marathe, M. Marathe, and S. Thite. 2004. Marketecture: A simulation-based framework for studying experimental deregulated power markets. In *Proceedings of the 6th IAEE European Energy Conference*. To appear.

Barrett, C., R. Beckman, K. Berkbigler, K. Bisset, B. Bush, K. Campbell, S. Eubank, K. Henson, J. Hurford, D. Kubicek, M. Marathe, P. Romero, J. Smith, L. Smith, P. Speckman, P. Stretz, G. Thayer, E. Eeckhout, and M. Williams. 2001. Transims: Transportation analysis and simulation system. Technical Report LA-UR-00-1725, Los Alamos National Laboratory.

Barrett, C., S. Eubank, V. Kumar, and M. Marathe. 2004, May. Understanding large-scale social and infrastructure networks. *SIAM News* 37 (4).

Barrett, C., H. Hunt, M. Marathe, S. Ravi, D. Rosenkrantz, and R. Stearns. 2003, February. Reachability problems for sequential dynamical systems with threshold functions. *Theoretical Computer Science* 295 (1–3): 41–64.

Barrett, C., H. Mortveit, and C. Reidys. 2000. Elements of a theory of simulation II: Sequential dynamical systems. *Applied Mathematics and Computation* 107:121–136.

Brand, D., and P. Zafiropulo. 1983. On communicating finite-state machines. *Journal of the ACM* 30 (2): 323–342.

Cho, H., and R. Wysk. 1995. Intelligent workstation controller for computer-integrated manufacturing: Problems and models. *Journal of Manufacturing Systems* 14 (4): 252–263.

Concepcion, A., and B. Zeigler. 1988. DEVS formalism: A framework for hierarchical model development. *IEEE Transactions on Software Engineering* 14 (2): 228–241.

Davis, W., C. Pegden, K. Musselman, R. Ingalis, and W. Trybula. 1991. Simulation and scheduling. In *Proceedings of the 1991 Winter Simulation Conference*, 382–391.

Dilts, D., N. Boyd, and H. Whorms. 1991. The evolution of control architectures for automated manufacturing systems. *Journal of Manufacturing Systems* 10 (1): 79–93.

Drake, G., J. Smith, and B. Peters. 1995. Simulation as a planning and scheduling tool for flexible manufacturing systems. In *Proceedings of the 1995 Winter Simulation Conference*, 805–812.

Eubank, S., H. Guclu, V. Kumar, M. Marathe, A. Srinivasan, Z. Toroczkai, and N. Wang. 2004. Modeling disease outbreaks in realistic urban social networks. *Nature* 429:180–184.

Jones, A., and A. Saleh. 1990. A multi-level/multi-layer architecture for intelligent shop floor control. *International Journal of Computer Integrated Manufacturing* 3 (1): 60–70.

Joshi, S., E. Mettala, J. Smith, and R. Wysk. 1995. Formal models for control of flexible manufacturing cells: Physical and system model. *IEEE Transaction on Robotics and Automation* 11 (4): 558–570.

Kasturia, E., F. DiCesare, and A. Desrochers. 1988. Real-time control of multilevel manufacturing systems using colored petri-nets. In *Proceedings of the IEEE International Conference on Robotics and Automation*, 1114–1119.

Li, M., and P. Vitányi. 1997. *An introduction to Kolmogorov complexity and its applications*. 2nd ed. Springer-Verlag.

Manivannan, S., and J. Banks. 1991. Real-time control of a manufacturing cell using knowledge-based simulation. In *Proceedings of the 1991 Winter Simulation Conference*, 251–260.

Marathe, M., H. Hunt, R. Stearns, and V. Radhakrishnan. 1998. Approximation algorithms for PSPACE-hard hierarchically and periodically specified problems. *SIAM Journal on Computing* 27 (5): 1237–1261.

McConnell, P., and D. Medeiros. 1992. Real-time simulation for decision support in continuous flow manufacturing systems. In *Proceedings of the 1992 Winter Simulation Conference*, 936–944.

Mettala, E. 1989. *Automatic generation of control software in computer integrated manufacturing*. Ph. D. thesis, University Park, Pennsylvania.

Radiya, A., and R. Sargent. 1987. A new formalism for discrete event simulation. In *Proceedings of the 1987 Winter Simulation Conference*, 554–558.

Radiya, A., and R. Sargent. 1994. A logic-based foundation of discrete event modeling and simulation. *ACM Transaction on Modeling and Computer Simulations*:3–51.

Smith, J., and S. Joshi. 1995. A shop floor controller class for computer-integrated manufacturing. *International Journal of Computer Integrated Manufacturing* 8 (5): 327–339.

Smith, J., R. Wysk, D. Sturrok, S. Ramaswamy, G. Smith, and S. Joshi. 1994. Discrete event simulation for shop floor control. In *Proceedings of the 1994 Winter Simulation Conference*, 962–969.

Son, Y., H. Rodríguez-Rivera, and R. Wysk. 1999. A multi-pass simulation-based, real-time scheduling and shop floor control system. *The quarterly Journal of the Society for Computer Simulation International* 16 (4): 159–172.

Srinon, R., and S. Ramakrishnan. 2005. Simulation modeling using neutral xml constructs. In *Proceedings of the International Conference on Systems Engineering*.

Wu, S., and R. Wysk. 1988. Multi-pass expert control system: A control/scheduling structure for flexible manufacturing cells. *Journal of Manufacturing Systems* 7 (2).

Wu, S., and R. Wysk. 1989. An application of discrete-event simulation to on-line control and scheduling in flexible manufacturing. *International Journal of Production Research* 27 (9): 1603–1623.

Zeigler, B., H. Praehofer, and T. Kim. 2000. Theory of modeling and simulation.

AUTHOR BIOGRAPHIES

SREERAM RAMAKRISHNAN is an assistant professor of Engineering Management and Systems Engineering in the University of Missouri–Rolla. His research interests include simulation modeling of distributed systems, simulation-based control, and hierarchical modeling for control systems. He is a member of IIE and ASEE. His e-mail address is <sreeram@umr.edu> and his web address is <http://www.umr.edu/~sreeram>.

MAYUR THAKUR is an assistant professor of computer science at the University of Missouri–Rolla. His research interests include network and graph algorithms, computational complexity theory, theory of discrete simulations, and quantum computing. He is an associate editor for the Journal of Universal Computer Science. His e-mail address is <thakurk@umr.edu> and his web address is <http://www.umr.edu/~thakurk>.