MODELING OF DISCRETE EVENT SYSTEMS

Stéphane Lafortune
The University of Michigan, USA

Keywords: Automaton, observer, parallel composition, Petri net, process algebra, product, regular languages.

Contents

Summary

The modeling formalism of automata is considered and the manipulation of automata for the construction of complete system models from individual component models is discussed. These manipulations include the operations of product and parallel composition. The notion of blocking in the context of language models and automaton models is presented. Blocking captures the phenomena of deadlock and livelock that can occur in the behavior of discrete event systems. An algorithm for the construction of observer automata from automata with unobservable transitions is presented. The modeling formalisms of Petri nets and communicating sequential processes are also treated and contrasted to automata in the context of a simple resource sharing example.

1. Introduction

Discrete event systems are dynamic systems with discrete state spaces and event-driven

dynamics. The area of discrete event systems is multidisciplinary and involves concepts and techniques from computer science theory, control theory, and operations research. A wide variety of modeling formalisms is being used to describe and study the behavior of discrete event systems. Two widely-used modeling formalisms in control engineering are *automata* and *Petri nets*. A lot of progress has been made in the last two decades in the development of a control theory for discrete event systems modeled as automata or Petri nets. For this reason, these are the formalisms discussed in this chapter; in addition, a brief discussion of *communicating sequential processes* is given in Section 6.

Discrete event models of dynamic systems are classified in terms of how they abstract timing information and randomness in the system behavior. Untimed models abstract away precise timing issues by focusing only on the ordering of the events and not on the exact times of their occurrence. Untimed models also abstract away statistical information about the probabilities of the events and consider all possible "sample paths" in the system behavior. Untimed models are often referred to as "logical" models. Timed models enrich untimed models and explicitly include timing information. This information may be provided in a "deterministic" manner or in a "stochastic" manner. The process of model refinement from untimed automata to stochastic timed automata and their associated generalized semi-Markov stochastic processes are discussed in *Discrete Event Systems*.

This chapter explores in further depth discrete-event modeling by building on the discussion in *Discrete Event Systems*. The primary focus of this chapter is untimed models of discrete event systems. Deterministic and stochastic timed models are briefly discussed in *Discrete Event Systems*; stochastic timed models are also considered in *Sample Path Analysis of Discrete Event Dynamic Systems*.

1.1. Formal Languages

The concept of *language* is introduced in *Discrete Event Systems* for modeling the logical behavior of a discrete event system. An untimed language, or simply language, is a set of strings of events over an event set. Let E be the finite set of events (or "alphabet") associated with the discrete event system under consideration. This set consists of all the events that can possibly be executed by the system. A *string* (or trace) is a finite sequence of events from E. The length of a string s, denoted by $|s|$, is a non-negative integer corresponding to the number of events composing the string, counting multiple occurrences of the same event. The empty string, denoted by ε (not to be confused with the generic event $e \in E$), is the string containing no events, i.e., $|\varepsilon| = 0$. The concatenation of two strings s_1 and s_2 is the trace $s_1 s_2$ (i.e., s_1 followed by s_2). Thus the empty string ε can be interpreted as the identity element for concatenation.

Denote by E^* the set of all finite strings of elements of E, including the empty string ε; the * operation is called the Kleene closure. For example, if $E = \{a, b, c\}$, then

$$E^* = \{\varepsilon, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, ...\}.$$

A *language* is then formally defined as a *subset* of E^*. If $s't = s$ with $s, s', t \in E^*$, then s' is called a *prefix* of s and t a *suffix* of s. Both ε and s are prefixes of s, by definition.

The usual set operations, such as union, intersection, difference, and complement (with respect to E^*) are applicable to languages since languages are sets. In addition, consider the following operations (":=" denotes "equal to by definition"):

- *Concatenation*: Let $L_1, L_2 \subseteq E^*$, then

$$L_1 L_2 := \left\{ s \in E^* : (s = s_1 s_2) \wedge (s_1 \in L_1) \wedge (s_2 \in L_2) \right\}.$$

- *Prefix-closure*: Let $L \subseteq E^*$, then

$$\overline{L} := \left\{ s \in E^* : \exists t \in E^* (st \in L) \right\}.$$

Thus the prefix-closure $\overline{L}$ of L is the language consisting of all the prefixes of all the strings in L. In general, $L \subseteq \overline{L}$. L is said to be *prefix-closed* if $L = \overline{L}$.

2. Automata

2.1. Basic Concepts

The automaton modeling formalism is introduced in *Discrete Event Systems*. For the sake of completeness, the definition of an automaton is recalled.

Automaton: A *deterministic automaton*, denoted by G, is a six-tuple

$$G = (X, E, f, \Gamma, x_0, X_m)$$

where: X is the set of *states*, which could be infinite; E is the finite set of *events* associated with the transitions in G; $f : X \times E \to X$ is the *transition function*: $f(x,e) = y$ means that there is a transition labeled by event e from state x to state y (in general, f is a *partial* function on its domain); $\Gamma : X \to 2^E$ is the *feasible event function*: $\Gamma(x)$ is the set of all events e for which $f(x,e)$ is defined; x_0 is the *initial* state; $X_m \subseteq X$ is the set of *marked states*. (Given a set A, the notation 2^A means the power set of A, i.e., the set of all subsets of A.) Proper selection of which states to mark is a modeling issue that depends on the problem of interest. By designating certain states as marked, we may for instance be recording that the system, upon entering these states, has completed some operation or task (cf. example in Section 3.2).

It is convenient to represent graphically automata where $|X|$ is finite and small by means of their *state transition diagrams*. The state transition diagram of an automaton is a directed graph where nodes represent states and labeled arcs between nodes are used

to represent the transition function f: if $f(x,e) = y$, then an arc labeled by "e" is drawn from x to y. Special notation is used to identify the initial states and marked states. In Figs. 2-4 below, the initial state is identified by an arrow pointing into it and marked states are differentiated by means of a double circle or box.

For the sake of convenience, the transition function f of an automaton is extended from domain $X \times E$ to domain $X \times E^*$ in the following recursive manner:

$$f(x,\varepsilon) := x$$
$$f(x,se) := f(f(x,s),e) \text{ for } s \in E^* \text{ and } e \in E .$$

2.2. Languages Represented by Automata

Automata are used to represent and manipulate languages. The *language generated* by $G = (X, E, f, \Gamma, x_0, X_m)$ is

$$L(G) := \{s \in E^* : f(x_0, s) \text{ is defined}\} .$$

The *language marked* by G is

$$L_m(G) := \{s \in L(G) : f(x_0, s) \in X_m\} .$$

The language $L(G)$ represents all the directed paths that can be followed along the state transition diagram of G, starting at the initial state; the string corresponding to a path is the concatenation of the event labels of the transitions composing the path. Therefore, a string s is in $L(G)$ if and only if it corresponds to an admissible path in the state transition diagram, equivalently, if and only if f is defined at (x_0, s). $L(G)$ is prefix-closed by definition, since a path is only possible if all its prefixes are also possible. If f is a total function over its domain, then necessarily $L(G) = E^*$.

The second language represented by G, $L_m(G)$, is the subset of $L(G)$ consisting only of the strings s for which $f(x_0, s) \in X_m$, i.e., these strings correspond to paths that end at a marked state in the state transition diagram. Since not all states of X need be marked, the language marked by G, $L_m(G)$, need not be prefix-closed in general. The language marked is also called the language *recognized* by the automaton, and the given automaton is often referred to as a *recognizer* of the given language.

An automaton G thus represents *two* languages: $L(G)$ and $L_m(G)$. In the standard definition of automaton in automata theory, the function f is required to be a total function and the notion of language generated is not meaningful since it is always equal to E^*. Allowing f to be partial is a consequence of the fact that a discrete event system may not be able to produce (or execute) all strings in E^*.

2.2.1. Blocking: Deadlock and Livelock

The definitions of G, $L(G)$, and $L_m(G)$ imply that in general

$$L_m(G) \subseteq \overline{L_m(G)} \subseteq L(G),$$

since X_m may be a proper subset of X. It is worth examining the second set inclusion in more detail.

An automaton G could reach a state x where $\Gamma(x) = \varnothing$ but $x \notin X_m$. This is called a *deadlock* because no further event can be executed. Given the interpretation of marking, this means that the system "blocks" because it enters a deadlock state without having terminated the task at hand. If deadlock happens, then necessarily $\overline{L_m(G)}$ will be a proper subset of $L(G)$, since any string in $L(G)$ that ends at state x cannot be a prefix of a string in $L_m(G)$.

Another issue to consider is when there is a set of unmarked states in G that forms a strongly connected component (i.e., these states are reachable from one another), but with *no transition going out of the set*. If the system enters this set of states, then *livelock* results. While the system is "live" in the sense that it can always execute an event, it can never complete the task started since no state in the set is marked and the system cannot leave this set of states. If livelock is possible, then again $\overline{L_m(G)}$ will be a proper subset of $L(G)$. Any string in $L(G)$ that reaches such an "absorbing" set of unmarked states cannot be a prefix of a string in $L_m(G)$, since it is assumed that there is no way out of this set. Again, the system is "blocked" in the livelock. The importance of deadlock and livelock in discrete event systems leads to the following definition.

Blocking: Automaton G is said to be *blocking* if $\overline{L_m(G)} \subset L(G)$, where the set inclusion is proper, and *nonblocking* when $\overline{L_m(G)} = L(G)$. Thus, if an automaton is blocking, this means that deadlock and/or livelock can happen.

The notion of marked states and the definitions of language generated, language marked, and blocking, provide an approach for considering deadlock and livelock that is useful in a wide variety of applications.

2.3. Accessibility Properties

From the definitions of $L(G)$ and $L_m(G)$, all the states of G that are not *accessible* or *reachable* from x_0 by some string in $L(G)$ can be deleted without affecting the languages generated and marked by G. When "deleting" a state, all the transitions that are *attached* to that state are also deleted. This operation is denoted by $Ac(G)$, where Ac stands for taking the "accessible" part.

A state x of G is said to be *coaccessible to* X_m, or simply coaccessible, if there is a path in the state transition diagram of G from x to a marked state. The operation of deleting all the states of G that are *not* coaccessible is denoted by $CoAc(G)$, where $CoAc$ stands for taking the "coaccessible" part. Taking the coaccessible part of an automaton means building

$$CoAc(G) := (X_{coac}, E, f_{coac}, x_{0,coac}, X_m) \quad \text{where}$$

$$X_{coac} = \left\{ x \in X : \exists s \in E^* \ (f(x,s) \in X_m) \right\}$$

$$x_{0,coac} = \begin{cases} x_0 & \text{if } x_0 \in X_{coac} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$f_{coac} = f|_{X_{coac} \times E \to X_{coac}},$$

where the notation $f|_{X_{coac} \times E \to X_{coac}}$ means the restriction of function f to domain $X_{coac} \times E$. The $CoAc$ operation may shrink $L(G)$, since states that are accessible from x_0 may be deleted; however, the $CoAc$ operation does not affect $L_m(G)$, since any deleted state is not on any path from x_0 to X_m. If $G = CoAc(G)$, then G is said to be *coaccessible*; in this case, $L(G) = \overline{L_m(G)}$. Coaccessibility is closely related to the concept of blocking. Blocking necessarily means that $\overline{L_m(G)}$ is a proper subset of $L(G)$ in which case there are accessible states that are not coaccessible.

Note that if the *CoAc* operation results in $X_{coac} = \varnothing$ (this would happen if $X_m = \varnothing$ for instance), then the *empty automaton* is obtained. The term "empty automaton" refers to an automaton whose state space is empty; an empty automaton necessarily generates and marks the empty set.

An automaton that is both accessible and coaccessible is said to be *trim*. The *Trim* operation is defined as follows:

$$Trim(G) := CoAc[Ac(G)] = Ac[CoAc(G)]$$

where the commutativity of Ac and $CoAc$ is easily verified.

2.4. Nondeterministic Automata

In the definition of automaton, state transitions describe how an event e causes a transition from some state x to a unique new state y. Suppose, however, that an event e at state x may cause transitions to more than one new states. This is a useful manner of modeling randomness in the system behavior. In this case, $f(x,e)$ should no longer represent a specific new state $x \in X$, but rather a *set* of possible new states. In addition, let the label ε be allowed in the state transition diagram of an automaton, i.e., allow transitions between distinct states to have the empty string as label. These transitions may represent events that cause a change in the "internal" state of the system but are not "observable" by an outside observer – imagine that there is no sensor that records this state transition. Thus the outside observer cannot attach an event label to such a

transition but it recognizes that the transition may occur by using the ε label. These two changes lead to the notion of a *nondeterministic automaton*.

Nondeterministic automaton: A *nondeterministic automaton*, denoted by G_{nd}, is a six-tuple

$$G_{nd} = (X, E \cup \{\varepsilon\}, f_{nd}, \Gamma, x_0, X_m)$$

where these objects have the same interpretation as in the definition of deterministic automaton, with the two differences that:

1. f_{nd} is a function $f_{nd} : X \times (E \cup \{\varepsilon\}) \to 2^X$, i.e., $f_{nd}(x,e) \subseteq X$ whenever it is defined.
2. The *initial* state may itself be a set of states, i.e., $x_0 \subseteq X$.

3. Operations on Automata

Discrete event models of complex dynamic systems are rarely built in a monolithic manner. Instead, a modular approach is used where models of individual components are built first, followed by the composition of these models in order to obtain the model of the overall system. The synchronization, or coupling, between components can be captured by the use of *common events* between system components. Namely, if components A and B share event c, then event c should only occur if both A and B execute it. The process of composing individual automata (that model interacting system components) in a manner that captures the synchronization constraints imposed by their common events is formalized by the *product* and *parallel composition* operations that are studied in this section.

-
-
-

Bibliography

Alur R. and Dill, D.L. (1994). *A Theory of Timed Automata*. Theoretical Computer Science. **126**, pp. 183–235. [This paper presents a timed discrete-event modeling formalism that has been studied extensively in the computer science literature.]

Arnold A.(1994). *Finite transition systems*. International Series in Computer Science. Prentice-Hall. [A more advanced book on untimed modeling of discrete event systems.]

Cassandras C.G. and Lafortune S. (1999). *Introduction to Discrete Event Systems*. Kluwer Academic Publishers. [Chapters 2 and 4 of this textbook on discrete event systems present a detailed coverage of automata and Petri net models.]

David R. and Alla H. (1992). *Petri Nets and Grafcet: Tools for Modelling Discrete Event Systems*. Prentice-Hall. [A nice textbook on Petri net models and their connection with Grafcet, a programming method for programmable logic controllers (PLCs), which are widely-used in industrial automation.]

Harel D. and Politi M. (1998). *Modeling Reactive Systems with Statecharts: The Statemate Approach*. Wiley. [This book describes a class of hierarchical automaton models that has become popular in commercial software packages for modeling discrete-event dynamics.]

Hoare C.A.R. (1985). *Communicating Sequential Processes*. International Series in Computer Science. Englewood Cliffs, NJ, Prentice-Hall. [A textbook treatment of the modeling formalism of communicating sequential processes.]

Hopcroft J.E. and Ullman J.D. (1979). *Introduction to Automata Theory, Languages, and Computation*. Reading, MA, Addison-Wesley. [A classic textbook on automata and language theory.]

Kurshan R.P. (1994). *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press. [A book on a class of automaton models in the context of formal verification.]

Manna Z. and Pnueli A. (1992). *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag. [This book describes the use of temporal logic in the specification and verification of timed discrete event systems.]

Murata T. (1989). *Petri Nets: Properties, Analysis, and Applications*. PIEEE, Vol. 77, **4**, Apr, pp.541–580. [An excellent tutorial/survey paper on Petri net models.]

Biographical Sketch

Stéphane Lafortune received the B. Eng degree from École Polytechnique de Montréal in 1980, the M. Eng. degree from McGill University in 1982, and the Ph.D. degree from the University of California at Berkeley in 1986, all in electrical engineering. Since September 1986, he has been with the University of Michigan, Ann Arbor, where he is a Professor of Electrical Engineering and Computer Science. Dr. Lafortune is a Fellow of the IEEE (1999). He received the Presidential Young Investigator Award from the National Science Foundation in 1990 and the George S. Axelby Outstanding Paper Award from the Control Systems Society of the IEEE in 1994 (for a paper co-authored with S. L. Chung and F. Lin) and in 2001 (for a paper co-authored with G. Barrett). Dr. Lafortune was a member of the Editorial Board of the Journal of Discrete Event Dynamic Systems: Theory and Applications in the period 1993-2000. He was Associate Editor at Large (1996-1999) and Associate Editor (1993-1996) of the IEEE Transactions on Automatic Control. His research interests are in discrete event systems, intelligent transportation systems, and communication networks. He co-authored, with C. Cassandras, the textbook *Introduction to Discrete Event Systems* (Kluwer Academic Publishers, 1999). Recent publications are available at the Web site www.eecs.umich.edu/umdes.