Control of Dynamic Discrete-Event Systems

by

Lenko Grigorov Grigorov

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Master of Science

Queen's University
Kingston, Ontario, Canada
January 2004

Abstract

In this work a type of discrete-event systems which vary with time, named *dynamic discrete-event systems*, is defined and a method for the control of such systems is proposed. Two major topics are discussed: redundancy structures for the efficient reconstruction of the complete system models when constituent modules change, and the use of online control to optimize the supervision of dynamic systems. Three redundancy structures are proposed: *stack redundancy*, *tree redundancy* and *hybrid redundancy*. The control algorithms are designed to address the specific requirements of the class of systems considered. The *online control* paradigm is used since it can adapt automatically to the changes that occur in dynamic systems. A *value function* is used to guide the supervision of systems, allowing for optimal control and for a flexible way to set requirements on the expected system behavior. This function, combined with the definition of *goals*, is shown to successfully replace marking in automata and to support the work of continuous-life systems using infinite goals. It is shown experimentally that the proposed modifications offer a significant improvement in the quality of control of dynamic systems. The results of this work can be applied to the control of large dynamic systems with prioritizing: such as dispatching centers, factories, resource access managers, and others.

Acknowledgments

I would like to thank my supervisor, Dr. Karen Rudie, for being a wonderful person and for helping me much more than what her position calls for.

I would like also to thank Dr. Kai Salomaa who acted professionally and supported me at a critical point in the thesis submission process.

I am grateful to Dr. R. Browse, Dr. K. Hashtrudi-Zaad, Dr. H. Hassanein and Dr. K. Salomaa for their helpful comments during my defence.

I appreciated the help and support of many other people, including Arezou, Jonathan, Mrs. Robertson, my family and friends.

My work would not have been possible without the financial support I received from my supervisor and the university.

Contents

List of Tables

List of Figures

Chapter 1

Problem description

Discrete-Event Systems Control is a new and exciting field of research. There are numerous ways one can model discretely a system which reacts to or generates events. The research on control of such systems, though, usually revolves around the standard supervisory control theory (SSCT), as proposed by Ramadge and Wonham [23]. This theory is well-studied and has been adjusted by scientists to fit a number of different requirements or situations.

In this work I will focus on a solution for the control of a specific class of discrete-event systems. I will consider systems which are dynamic (change with time), which are relatively large, which have a continuous lifecycle (i.e., they continuously execute tasks from a set of tasks), and for which the users might wish to set requirements with different levels of stringency. These properties are natural for many systems found in real life. For example, a computer operating system might be executing different applications, which appear or terminate, which have different priority settings, etc. Another example is a truck-dispatching center, where different trucks are available at different times and where different tasks have different levels of importance.

Real-life discrete-event systems tend to be large. This is mostly because physical behavior of systems is (at least at human-perceivable scale) continuous. As long as the granularity with which we model increases, so does the complexity of the model. This is an obvious result. However, what causes a problem here is the nature of this relation: linear increase in granularity may result in exponential increase in complexity. This is true even for systems which consist of a number of components (or modules). While the complexity of a single module is very low, combining many of these has a dramatic effect on the complexity of the overall system. Thus, it is reasonable to expect that a controller for real-world discrete-event systems will have to deal with large systems.

Another aspect of many real-world systems is their uncertainty. While in a well-designed and tightly-controlled factory most of the processes are predictable (with the exception of occasional breakdowns), when the system involves or models interactions between disparate entities (especially when this involves humans), the predictability of processes decreases substantially. Even when modeling the highly predictable internal factory processes, one might choose to model also the dependence on external supplies. Since the factory management has no control over processes outside of the factory, it is no longer possible to blindly rely on a constant and timely flow of supplies. The behavior of parts of the system becomes uncertain and this needs to be taken into account during the control. In this work I consider a specific type of system dynamics—the appearance or disappearance of a group of modules at different points during the functioning of the system. While restrictive, this model can be applied to many systems, since the uncertainty usually is associated with the behavior of particular entities.

Control of a system is applied when the natural behavior of the system does not satisfy our requirements. Most of the time the problem is that the system may violate a requirement on its behavior. For example, a robot may continue dumping manufactured parts into a container even when the container gets full. The standard supervisory control theory was developed around the idea of having a system supervisor which makes sure such problems do not occur. A subset of the system behavior is delimited as being legal (admissible) and the supervisor prevents the system from exhibiting behavior outside of this subset. Such a solution presumes the world is black and white; and this certainly has applications, for example, in nuclear reactor control, space exploration, contingency reaction systems, etc. However, in most real-world systems there is a second requirement which is just as important. Not only would we like to prevent unwanted behavior, but we would also like to encourage desirable behavior. After all, systems are used to achieve some goals. While the robot might not be doing anything wrong by not dumping parts at all, one would like parts to be dumped while the container is not full. Thus, a refinement of the legal behavior is needed. The controlled system will be guided toward the execution of a task.

The discussion in this work will be concerned mainly with systems which are expected to function for a long time (have an infinite lifespan), such as operating systems, database systems, process co-ordinating systems, etc. The continuous lifecycle relates to the repetition of attempts to achieve a goal from a given set of goals. After one goal is achieved, the system starts working toward the achievement of another goal. While the results will be applicable to short-lived systems as well, depending on the size of these systems, SSCT-based solutions or the method proposed in [10] might achieve better performance.

The contributions of this work are as follows:

- The definition of a class of Discrete-Event Systems, called Dynamic Discrete-Event Systems (DDES), which can be used to model the aforementioned systems. The formal definition is presented in Chapter 4.

- The definition of three types of redundancy structures which can be used to speed up the process of recomputing of large systems consisting of separate modules. Algorithms for the creation and maintenance of such structures are also developed. The results are presented in Chapter 5.

- The development of a new method for the specification of requirements for the behavior of Discrete-Event Systems, using a *value* and a *goal* function. This method is shown to be more suitable for use with DDESs and it allows for specifications with different level of stringency. The functions are discussed in Sections 6.1 and 6.2.

- The modification of the online control algorithm, [7], to achieve near-optimal control of DDESs using the value and goal functions. The algorithms for the new control method are presented in Chapters 6 and 7. A proof-of-concept simulation shows that the optimal control algorithm achieves better performance than the original online control algorithm, however, at the cost of increased computational complexity. The results of the simulation are presented in Chapter 8.

Chapter 2

Background

The introduction of digital computers in the past century has influenced human society beyond all initial expectations. However, many people do not notice that science and research have been affected in a similar way. Advertisement logos like "for your digital lifestyle", "DIGITall[1]", etc. are successful with users, because the most modern digital equipment has proved to outperform older "analog" machines. Yet science has been "digitally" influenced as well. Even though Discrete Mathematics, Information Theory, Formal Grammars and other fields have been developed before the creation of the first digital computers, they have gained in popularity and importance only through their reinvention as "Computer Science". In other cases, completely new areas of research have come into existence due to the digitalization of our world—such as the Control of Discrete-Event Systems.

[1]Trademark property of Samsung Electronics

2.1 Discrete-Event Systems

The discretization of calculations in a computer has served as an inspiration to many researchers. It was not long before people realized that many systems (especially digital systems) could be successfully modeled as Discrete-Event Systems (DES). Such systems are those where events (changes of state) happen spontaneously, are logically ordered relative to each other, and are not tied to a continuous global time.

An example of a DES is the high-level model of a vending machine. The machine has a number of discrete states, defined by how many articles are available inside the machine and how many coins are inserted. A change of state happens when a coin is inserted or when the machine delivers an article. These changes of states are named "events". Some events can happen only in a given state. For example, the machine will not deliver an item if there are no goods loaded, or if the correct amount of money is not inserted. Naturally, DES models can be applied to much more complex systems, such as manufacturing cells [16].

Discrete-Event Systems can be formally modeled using many different approaches, ranging from Petri nets to Markov chains, fuzzy matrixes [17], and modal logic [25, 22]. However, the most commonly used method is the representation through automata, and for all practical purposes—Finite-State Machines (FSM). Besides being a very intuitive approach, this also allows for the application of results from Automata Theory, which is a well-studied area.

An FSM is a five-tuple $G = (\Sigma, Q, \delta, q_0, Q_f)$, where Σ is a finite set of symbols (and is often called the *alphabet*), Q is a finite set of states, δ is a partial transition function $\Sigma \times Q \to Q$, q_0 is the initial state of the system, and $Q_f \subseteq Q$ is a subset of the states, which are defined to be "final" (a final state is sometimes also referred to

as a "marked state"). The special "empty" symbol ϵ, which does not belong to Σ, is used to denote the empty string (i.e., the string of length zero). The notation Σ^* stands for the set of all strings of symbols from Σ and ϵ. The transition function δ can be naturally extended to the partial function $\delta' : \Sigma^* \times Q \to Q$, where $\delta'(\sigma, q) = \delta(\sigma, q)$ for all $\sigma \in \Sigma$ and $q \in Q$ and $\delta'(\sigma s, q) = \delta'(s, \delta(\sigma, q))$ for $s \in \Sigma^*$, $\sigma \in \Sigma$, and $q \in Q$. Usually, δ' is denoted by δ and is used instead of the original transition function. An FSM can be interpreted as a DES if states are considered to be states of the system and transitions labeled with symbols from Σ are considered to be events happening in the system. Strings of symbols would describe sequences of events.

The language $L(G)$ is defined to be the set of all possible sequences of events in the system. The FSM G is said to generate $L(G)$. The language $L_m(G)$ is defined to be the set of all sequences of events which lead to a final state. The FSM G is said to accept $L_m(G)$. The generated language $L(G)$ is always a superset of $L_m(G)$. More formally,

$$L(G) = \{s \mid s \in \Sigma^*, \delta(s, q_0) \text{ is defined}\},$$

$$L_m(G) = \{s \mid s \in \Sigma^*, \delta(s, q_0) \text{ is defined}, \delta(s, q_0) \in Q_f\},$$

$$\text{and } L_m(G) \subseteq L(G).$$

The *prefix-closure* of a language is defined to be the set of all prefixes of strings in the language. The empty string ϵ is a prefix of any string. For all automata, prefix-closing the generated language produces a language equal to the generated language itself. More formally,

$$\overline{L} = \{s \mid s \in \Sigma^*, \exists t \in \Sigma^*, st \in L\},\ \overline{L(G)} = L(G).$$

A *prefix-closed language* is a language which equals its prefix-closure. Prefix closure

is an important operation because it describes all the possible partial behaviors of a DES. An example of a DES is the simplified model of a customer at a store (Fig. 2.1).

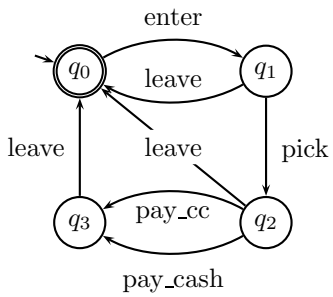


Figure 2.1: DES model of a customer in a store

The customer can enter the store, pick something to buy, pay with cash or a credit card, and leave at any time. Here $\Sigma = \{$"enter", "pick", "pay_cash", "pay_cc", "leave"$\}$. The set of states is $Q = \{q_0, q_1, q_2, q_3\}$. The transition function can be determined from the diagram in Fig. 2.1, e.g., $\delta(\text{pick}, q_1) = q_2$. The initial state is marked with q_0. This state is the only final state, as well (i.e., $Q_f = \{q_0\}$). Examples of event sequences are "enter, leave" or "enter, pick, pay_cc". The second sequence is not "complete"—it does not belong to L_m. However, it belongs to $\overline{L_m}$, since it is a prefix of the sequence "enter, pick, pay_cc, leave", which is in L_m.

The examples presented here are very simple, but one can easily imagine the application of DESs in factory processes, computer protocols, and other areas. This is why scientists are increasingly becoming interested in the DES paradigm.

2.2 Control of DES

Ever since the first mechanical machines have been built, their control has played a central role in the design. This is why Control Theory is a very important subject of Engineering. Classical Control Theory deals with the control of machines in continuous real time—employing, for example, differential equations. In the early 1980s researchers discovered the potential of describing systems as DESs. Naturally, one would not like to use DESs without having any means of control. Along with the many advantages that the discretization of a system's behavior brings, such as simplification and abstraction, there come also the issues of not having a well-developed theory. The pioneers of the Control Theory of DES are Ramadge and Wonham [23, 24].

How would one control a DES? As with any other system, the first thing to do is to specify the requirements on the system. This could be as simple as saying "the customer should not leave without paying". Of course, before being able to execute the requirements, the verbal specifications have to be formalized. The example above can be translated to "the sequence 'enter, pick, leave' must not happen in the system". If there are many requirements, we will end up with a list of sequences which must not happen in the system, and another one with sequences that have to happen. The above can be summarized easily by defining a sublanguage K of the accepted language L_m, such that K contains all and only the sequences that are desirable. This language is dubbed the "legal language", because all sequences outside of K are considered forbidden, i.e., they must not happen. Thus the specification of requirements translates to simply defining a legal language.

Before one starts exercising control, the limitations of the system have to be explored. As with other types of systems, not all parts of a DES need be controllable.

The alphabet Σ is thus partitioned into two complementary subsets, Σ_c and Σ_{uc}, the set of controllable events and the set of uncontrollable events, respectively.

$$\Sigma = \Sigma_c \cup \Sigma_{uc},\ \Sigma_c \cap \Sigma_{uc} = \emptyset$$

The controllable events are events which can be enabled or disabled. When enabled, the event can happen in the system, given that the underlying automaton can generate it. When disabled, the event cannot happen in the system. The uncontrollable events cannot be disabled and they may occur at any point, when they can be generated.

In the store-customer example, the set of controllable events can be $\Sigma_c = \{$"enter", "pay_cash", "pay_cc"$\}$ and the set of uncontrollable events can be $\Sigma_{uc} = \{$"pick", "leave"$\}$. Clearly, if the owner closes the store, customers will not be able to enter. The cashier can also control the method of payment. On the other hand, the behavior of the customers inside the store is not under control; customers can make their own decisions about whether they will pick something to buy or when to leave.

Once the requirements on the system are specified, and once the tools to control a DES are available, how does one exercise the control? There are two basic types of control: offline control and online control. The two approaches are discussed next.

2.3 Offline control

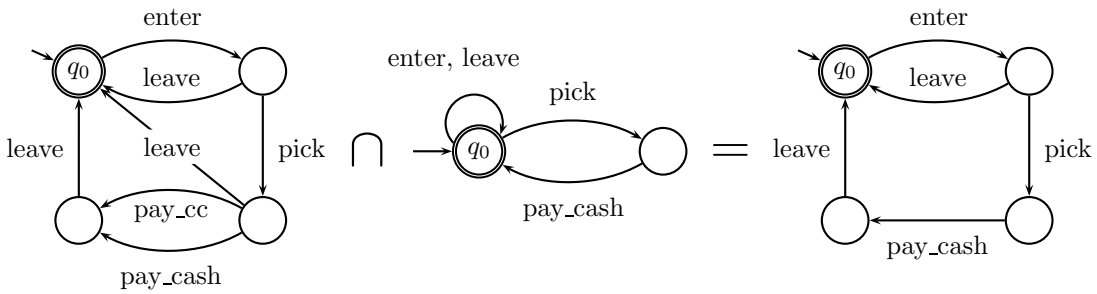
The offline control of DES was first proposed by Ramadge and Wonham [23]. Their work has served as the basis for most of the research on DES performed since then. Offline control uses a very simple approach. According to the requirement specifications, a new automaton is constructed. It is constructed in such a way that

it accepts only the legal language. Roughly speaking, the automaton describes when and which events have to be disabled, so that the requirements are met. The actual control occurs by intersecting the original DES automaton and the control automaton. This results in a system which behaves according to the requirements, i.e., it accepts the legal language. The intersection of two automata—$G_1 = (\Sigma_1, Q_1, \delta_1, q_{01}, Q_{f1})$ and $G_2 = (\Sigma_2, Q_2, \delta_2, q_{02}, Q_{f2})$—is defined as the automaton $G = (\Sigma_1 \cup \Sigma_2, Q_1 \times Q_2, \delta, [q_{01}, q_{02}], Q_{f1} \times Q_{f2})$, where the states are elements of the Cartesian product of the sets of states of the two automata, the transition function δ is defined as $\delta : (\Sigma_1 \cup \Sigma_2) \times (Q_1 \times Q_2) \rightarrow Q_1 \times Q_2$, $\delta(\sigma, [q_1, q_2]) = [\delta_1(\sigma, q_1), \delta_2(\sigma, q_2)]$ and is defined if and only if both $\delta_1(\sigma, q_1)$ and $\delta_2(\sigma, q_2)$ are defined. We use square brackets to denote the Cartesian product of states for better readability.

The approach is offline since it requires full knowledge of the DES and since the control automaton can be calculated once for the whole lifetime of the system. Due to the way the control automaton is applied, it is called a "supervisor" of the system, and the offline control is named "supervisory control of DES".

In the store-customer example already discussed, the legal language could be defined as $K = \{$"enter, leave", "enter, pick, pay_cash, leave"$\}$. One possible supervisor could look like Fig. 2.2(b). After intersecting the supervisor and the DES, the controlled system would look like Fig. 2.2(c). One can see how the new system can execute only sequences from the legal language. For example, the customer is limited in the payment methods he or she can use, i.e., only cash is accepted.

Being able to use supervisory control, however, does not guarantee that the restrictions can always be met. For example, since the "leave" event is uncontrollable, the supervisor cannot disable it. Thus, the customer will be able to leave after picking



(a) uncontrolled system (b) supervisor (c) controlled system

Figure 2.2: Offline supervisory control

something to buy and before he or she pays for the item, i.e., the controlled system could execute the sequence "enter, pick, leave" (Fig. 2.3).

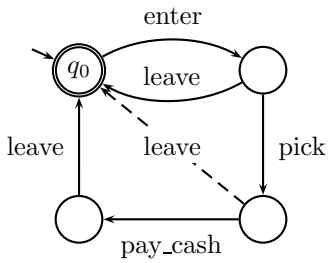


Figure 2.3: An uncontrollable event leads outside the legal language

2.3.1 Controllability and the supremal controllable sublanguage

Given that supervisors cannot control every event, is there a way to always ensure the DES behaves according to the legal language? As discussed in [4], this is not possible in all cases. Sometimes no supervisor can be constructed such that the supervised system executes only sequences from the legal language. The reason for this is that

from a sequence in the legal language an uncontrollable event can "shoot off", leading to a sequence outside the legal language. The sequence "enter, pick" is a prefix of the sequence "enter, pick, pay_cash, leave" which belongs to the legal language. However, the uncontrollable event "leave" may lead to the undesirable sequence "enter, pick, leave". Since, by definition, no supervisor can disable the "leave" event, a supervisor for the legal language cannot be constructed. This fact is formalized by the definition of controllability of a legal language with respect to a DES. A language is said to be controllable with respect to a DES if and only if no uncontrollable event leads outside the prefix closure of the legal language.

$$K \text{ is controllable with respect to } G \Leftrightarrow \{s\sigma \mid s \in \overline{K}, \sigma \in \Sigma_{uc}, s\sigma \in L(G)\} \subseteq \overline{K}$$

What could one do if the desired legal language is not controllable with respect to the DES? Controllability is always tied to a specific legal language; it is not an intrinsic property of a system. For two different legal languages K_1 and K_2, K_1 might not be controllable with respect to the system and K_2 might be controllable with respect to the system. Thus, if the legal language is modified, it might become controllable with respect to the system.

A very important observation on the properties of controllability has been made [29]. Controllability is preserved under union, that is, if both K_1 and K_2 are controllable with respect to G, then $K_1 \cup K_2$ is controllable with respect to G. Furthermore, $\emptyset$ is controllable with respect to any system. These results show that the class of all controllable sublanguages with respect to a DES, $\underline{C}(K, G) = \{L \mid L \subseteq K, L \text{ is controllable with respect to } G\}$, is a complete semilattice with a supremal element. This element usually is denoted by $\sup \underline{C}(K, G)$, the supremal controllable sublanguage of the legal language K.

Given the above results, if a legal language is not controllable with respect to the DES, one could consider the supremal controllable sublanguage instead. It is not necessary to choose the supremal controllable sublanguage, however, the reason behind this choice is that the supremal one matches most closely the original legal language. All other controllable languages impose greater restrictions on the system. Unfortunately, even the supremal controllable sublanguage may turn out to be empty. This is the case with the store-customer example. If "enter" is allowed to happen, the uncontrollable events "pick" and then "leave" might follow—which would lead outside the legal language. Thus "enter" has to be disabled. From here it follows that the supremal controllable sublanguage is the empty set.

In general, whether the legal language K is prefix-closed or not impacts significantly the complexity of algorithms which solve different problems. For example, the least-complex algorithm available for the computation of $\sup \underline{C}$ has a complexity of $O(mn|\Sigma|)$ for a prefix-closed legal language, and a complexity of $O(m^2n^2|\Sigma|)$ for the general case, where m and n are the number of states in the DES automaton and the supervising automaton respectively [4]. Since my work is interested in the control of dynamic DESs where "final" (or "marked") states do not always have meaning (see Section 6.1), I have chosen to consider only prefix-closed legal languages. All legal languages in the rest of this work are considered to be prefix-closed, unless stated otherwise.

The approach of offline control, or supervisory control, can be summarized as following:

1. Specify the requirements on the system.

2. Generate the legal language corresponding to the requirements.

3. Verify the controllability of the legal language with respect to the DES. If it is not controllable, generate the supremal controllable sublanguage of the legal language.

4. Generate the supervisor FSM for the language from step (3).

5. Intersect the supervisor and the DES to obtain the controlled system.

2.3.2 Modular control

As already mentioned, supervisory control is the most extensively studied type of control. Many modifications of the basic approach are proposed. These include control of modular DESs [30, 28, 8, 2], hierarchical DESs [32], distributed DESs [26], partially observable DESs [4] and others.

Modular DESs are of particular interest to my work because they serve to describe DESs as an assembly of modules. These modules are coupled using one of a number of possible compositions, of which the *parallel composition* (also called *synchronous product*) is the most widely used [4]. For two systems, $G_1 = (\Sigma_1, Q_1, \delta_1, q_{01}, Q_{f1})$ and $G_2 = (\Sigma_2, Q_2, \delta_2, q_{02}, Q_{f2})$, the synchronous product is defined to be the automaton $G_1 \| G_2 = (\Sigma_1 \cup \Sigma_2, Q_1 \times Q_2, \delta, [q_{01}, q_{02}], Q_{f1} \times Q_{f2})$, where the states are elements of the Cartesian product of the sets of states of the two automata, the transition function δ is defined as $\delta : (\Sigma_1 \cup \Sigma_2) \times (Q_1 \times Q_2) \rightarrow Q_1 \times Q_2$,

$$
\begin{aligned}
\delta(\sigma, [q_1, q_2]) &= [\delta_1(\sigma, q_1), \delta_2(\sigma, q_2)] \text{ if both } \delta_1(\sigma, q_1) \text{ and } \delta_2(\sigma, q_2) \text{ are defined,} \\
&= [\delta_1(\sigma, q_1), q_2] \text{ if only } \delta_1(\sigma, q_1) \text{ is defined and } \sigma \notin \Sigma_2, \\
&= [q_1, \delta_2(\sigma, q_2)] \text{ if only } \delta_2(\sigma, q_2) \text{ is defined and } \sigma \notin \Sigma_1, \\
&\quad \text{undefined otherwise.}
\end{aligned}
$$

In this work a modification of the synchronous product will be used because it allows for greater freedom in the behavior of parallel systems. We will call the new composition *synchronous shuffle*. The only difference is in the definition of the transition function δ:

$$\begin{aligned}
\delta(\sigma,[q_1,q_2]) &= [\delta_1(\sigma,q_1),\delta_2(\sigma,q_2)] \text{ if both } \delta_1(\sigma,q_1) \text{ and } \delta_2(\sigma,q_2) \text{ are defined,} \\
&= [\delta_1(\sigma,q_1),q_2] \text{ if only } \delta_1(\sigma,q_1) \text{ is defined,} \\
&= [q_1,\delta_2(\sigma,q_2)] \text{ if only } \delta_2(\sigma,q_2) \text{ is defined,} \\
&\quad \text{undefined if neither } \delta_1(\sigma,q_1) \text{ nor } \delta_2(\sigma,q_2) \text{ are defined.}
\end{aligned}$$

The synchronous shuffle of two DES modules simply means that the resultant system can execute the events that any one of the constituent systems can execute, and that the events are executed in both systems synchronously when possible.

If we consider the example discussed before, there could be two customers in the store. Each customer will have its own set of events ("enter$_1$", "leave$_1$"... and "enter$_2$", "leave$_2$"...) If the customers are a couple, they could pay together if they have chosen the items to buy, i.e., the "pay_cash" event will be common for both modules (Fig. 2.4). The result of the synchronous shuffle is shown in Table 2.1.

The modularity of DESs can be used to describe very complex systems as a parallel composition of a number of smaller and simpler systems. This not only helps people in being able to comprehend how a DES works, but also can be used to improve the performance of different algorithms. A successful application is the modular supervisory control [30]. Instead of creating one big supervisor for the complete DES, a separate "local" supervisor for each module in the system is designed. The supervisor for the complete system is then constructed by intersecting the two supervisors. This approach significantly reduces the complexity of calculations. For a prefix-closed legal

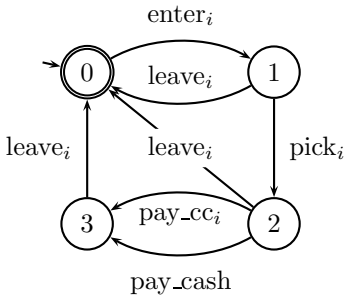
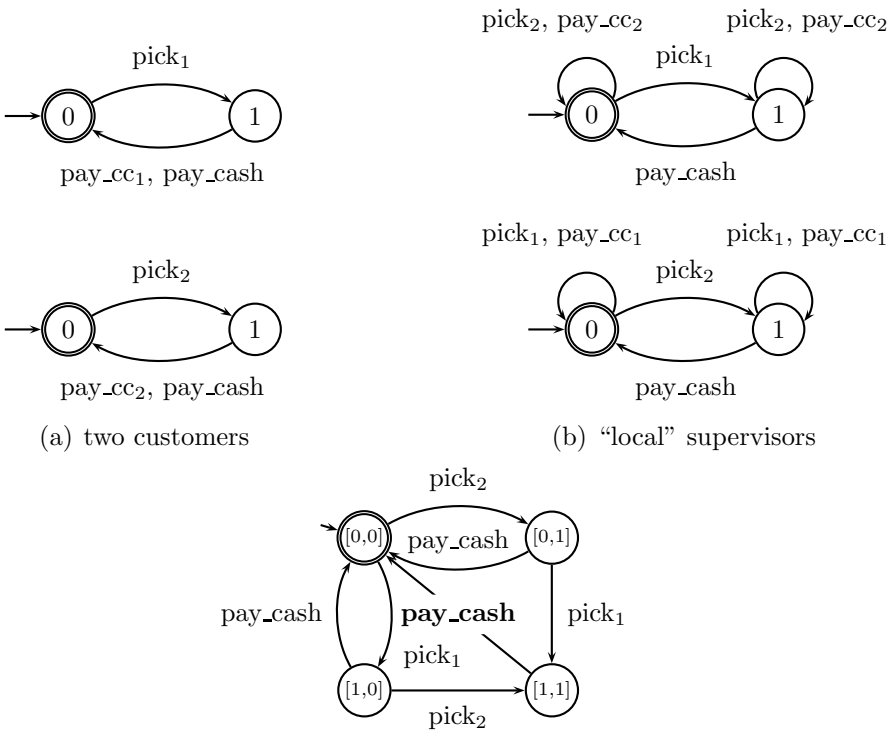


Figure 2.4: DES for two customers, $i \in \{1, 2\}$. The event pay_cash is a common event.

language, the complexity is reduced from $O(mn|\Sigma|)$ to $O(\max(m, n)|\Sigma|)$, where m, n are the number of states in the two modules.

An example of modular control is given in Fig. 2.5. Two separate supervisors for the two customers are constructed and then the complete system is supervised by the intersection of the supervisors. It is important to note that each "local" supervisor should always enable the "foreign" events, otherwise intersection will not produce the desired result.

When dealing with non-prefix-closed languages, the modular supervision would not always produce a correct result, since each "local" supervisor is not able to see the "big picture" of the whole system [4]. In [31] a number of different approaches to supervisor construction for distributed control are investigated, however the principles can also be applied to modular control, so this paper can be useful reading for modular supervision.



(a) two customers

(b) "local" supervisors

(c) global supervisor for the complete system

Figure 2.5: Modular supervision. The legal language is the prefix closure of {"pick$_1$, pay_cash", "pick$_2$, pay_cash", "pick$_1$, pick$_2$, pay_cash", "pick$_2$, pick$_1$, pay_cash"}. The DESs for the customers were edited to reduce the complexity.

Table 2.1: Transitions of Customer$_1$ $\|$ Customer$_2$. The system has $4^2 = 16$ states:

Initial state	Event	New state
[0,0]	enter$_1$	[1,0]
	enter$_2$	[0,1]
[0,1]	pick$_2$	[0,2]
	leave$_2$	[0,0]
	enter$_1$	[1,1]
[0,2]	pay_cc$_2$	[0,3]
	pay_cash	[0,3]
	leave$_2$	[0,0]
	enter$_1$	[1,2]
[0,3]	leave$_2$	[0,0]
	enter$_1$	[1,3]
[1,0]	pick$_1$	[2,0]
	leave$_1$	[0,0]
	enter$_2$	[1,1]
[1,1]	pick$_1$	[2,1]
	leave$_1$	[0,1]
	pick$_2$	[1,2]
	leave$_2$	[1,0]
[1,2]	pick$_1$	[2,2]
	leave$_1$	[0,2]
	pay_cc$_2$	[1,3]
	pay_cash	[1,3]
	leave$_2$	[1,0]
[1,3]	pick$_1$	[2,3]
	leave$_1$	[0,3]
	leave$_2$	[1,0]
[2,0]	pay_cc$_1$	[3,0]
	pay_cash	[3,0]
	leave$_1$	[0,0]
	enter$_2$	[2,1]

Initial state	Event	New state
[2,1]	pay_cc$_1$	[3,1]
	pay_cash	[3,1]
	leave$_1$	[0,1]
	pick$_2$	[2,2]
	leave$_2$	[2,0]
[2,2]	pay_cc$_1$	[3,2]
	pay_cash	*[3,3]*
	leave$_1$	[0,2]
	pay_cc$_2$	[2,3]
	leave$_2$	[2,0]
[2,3]	pay_cc$_1$	[3,3]
	pay_cash	[3,3]
	leave$_1$	[0,3]
	leave$_2$	[3,0]
[3,0]	leave$_1$	[0,0]
	enter$_2$	[3,1]
[3,1]	leave$_1$	[0,1]
	pick$_2$	[3,2]
	leave$_2$	[3,0]
[3,2]	leave$_1$	[0,2]
	pay_cc$_2$	[3,3]
	pay_cash	[3,3]
	leave$_2$	[3,0]
[3,3]	leave$_1$	[0,3]
	leave$_2$	[3,0]

2.3.3 Issues with offline control

Simple as it is, offline control seems to be a very promising tool for the manipulation of systems which can be modeled as DESs. Once the requirements are specified as a legal language, the rest of the steps can be completely automated. There are algorithms capable of generating FSMs from regular languages, algorithms for the generation of the supremable controllable sublanguage, and algorithms to create synchronous shuffle and intersection of automata. After the initial excitement over the new convenient paradigm for control, however, a number of problems were identified. These include the enormous state space complexity of relatively simple systems and the inappropriateness of the supervisory approach for dynamically changing systems.

One of the most disappointing implications of the DES paradigm is its state-space complexity. If continuous systems are modeled, they can be described using real functions. Thus with a single equation (or a very small number of equations) the system can be described to an arbitrarily small level of detail. However, if the same system is to be described using discrete states, the level of detail will always be finite—it depends on the "coarseness" of the description. If a simple function is to be described, say the function $y = x$ for $x \in [0, 1)$, then a two-state DES could be used, for example, to represent the states $y_1 = x$ for $x \in [0, 0.5)$ and $y_2 = x$ for $x \in [0.5, 1)$. If twice the detail is required, the system will need four states: $y_1 = x$ for $x \in [0, 0.25)$, $y_2 = x$ for $x \in [0.25, 0.5)$, $y_3 = x$ for $x \in [0.5, 0.75)$, and $y_4 = x$ for $x \in [0.75, 1)$. For three times the detail, eight states will be needed, and so on. In other words, a linear increase on the requirements for detail leads to an exponential increase in the number of states required to describe the system. How severe this problem is can be easily illustrated. If the store with customers is considered once again, one could

easily imagine that there can be, for example, seven customers in the store. Each customer is described using a highly simplified DES, consisting of only four states. How complex would the complete system be? Using the synchronous shuffle, the resultant DES could have up to 16,384 states! The large number is obtained for a store with seven customers and this is a small store according to most standards.

A system with 16,000 states can still be successfully manipulated using any modern computer. However, if real problems are considered, there can be 20 or more modules and each module can have 100 or more states. This results in a system which can possibly have 100^{20} or more states. Is there a computer capable of performing computations on this number of states in a reasonable amount of time? This result shows that control of DES seems to be inappropriate for almost all practical purposes. A solution to this problem might be made available through quantum computations [3] since quanta are capable of performing an exponential number of calculations in linear time. Unfortunately, the development of real quantum computers is still in its very early stages and most of the research remains purely theoretical.

The next problem, connected with supervisory control of DES, is related to the one discussed above. How would one implement the control of systems which vary with time? Such change could be the flow of customers in a store. When the store opens, it would start with no customers, then a varying number of customers will come in and leave at different points during the open hours. If the store manager wishes to use a controlled DES to model the operation of the store, he or she will not be able to employ the supervisory control approach. Since the manager does not know in advance the number of customers for any given time, he or she cannot build a supervisor for the system.

For small systems, this problem can be solved in a couple of ways:

- perform all computations over again each time the number of customers changes
- use modular control and each time only compose the required number of modules
- use precomputed models for every possible number of customers

As is easily seen, these approaches can work only when the state space of the system is very small and when the range of possible changes to the system is limited and predictable. If a supervisor for a very large system can be computed using a supercomputer for one year, but the system changes every three months, would the offline approach be feasible?

The practical limitations of the offline control of DES have led to the proposal of other ways to control such systems.

2.4 Online control

Soon after the introduction of offline (supervisory) control of DES, it was recognized that even though the method is "optimal" in the sense of providing the correct answer, closest to our expectations, it is not applicable in most practical situations. Offline control requires complete knowledge of the controlled DES, so that controllability can be verified or the supremal controllable sublanguage can be constructed. Unfortunately, most real systems are so complex that it is almost impossible to describe them directly as a DES. The modular approach has been developed, namely, the system is divided into logical components (modules) which cooperate (work in parallel).

The description and the control of DESs are thus greatly simplified. This approach, however, fails to circumvent the greatest obstacle in the application of DESs: the state-space complexity. With the linear increase of the system (e.g., the number of modules), an exponential "explosion" of the number of states is observed. Even when it is practically possible to calculate a supervisor, the process is so long that it cannot be used when the system changes frequently. An example of a system which can be successfully modeled as a DES is the concurrency manager of databases. Even though each transaction has a very limited set of states (usually three), there can be tens of concurrent transactions in a database. The expected performance makes it impossible to wait even one second for each decision of the manager. Looking for a way to solve the issues with offline control, researchers have come up with the idea of online control [6].

Unlike the offline control, this new approach does not depend on complete knowledge of the system. Control is exercised from outside of the system. A unit, called a *controller*, intercepts events as they happen in the DES and synchronously makes decisions about which events to enable or disable. A diagram of the process is shown in Fig. 2.6. This method lets the control be carried out "online", i.e., the control decisions are made as the system evolves.

How would the controller know which controllable events to enable and which to disable? Two approaches have been proposed in the literature: abstracted control [6] and control using state information [10]. The abstracted approach requires only knowledge of the language, generated by the DES ($L(G)$), and the legal language (K). No assumptions are made on how the DES is implemented and how it works (whether it is an automaton, a Petri net, etc.) The approach using state information works

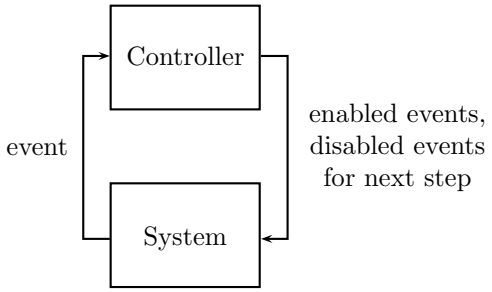


Figure 2.6: Online control scheme

only with FSM-implemented DESs and requires knowledge of the concrete structure of the system. Both approaches proceed by building what is called a "look-ahead window" and examining this structure to determine the control actions.

The operation of a DES consists of the occurrences of events. The events, which have happened in the system, create a sequence which describes the history of the system. After each event, there can be numerous ways a system can proceed. If we consider the store-customer example, after the sequence "enter", the system can proceed by executing either "leave" or "pick". After the sequence "enter, pick", the system can continue with "leave", "pay_cc", or "pay_cash". These possibilities create a tree-like structure (Fig. 2.7) which is called a *look-ahead window* or *tree*.

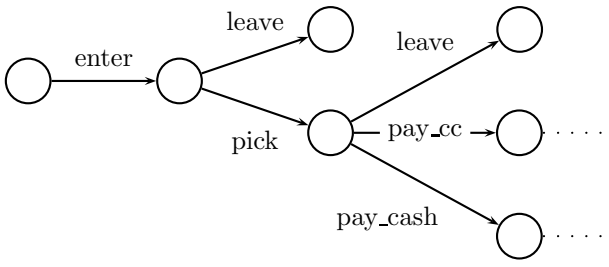


Figure 2.7: Look-ahead tree for the customer example

After the occurrence of an event, the controller has to decide on the enablement or disablement of the controllable events (since uncontrollable events remain always enabled). The decision should be such that the sequence, which will eventually be executed in the system, will belong to the legal language. Thus, examination of the look-ahead tree can provide the controller with the necessary information, since it contains the possible evolutions of the system. Each possible future sequence can be explored and if it leads to something outside of the legal language, the controller can decide to disable the controllable events leading to this sequence.

The use of the look-ahead tree is not very efficient, though, if no restrictions are imposed. If the controller can follow arbitrarily long potential sequences, each single iteration of the control process might end up having the same complexity as the calculation of the supervisor for the offline control. This would only increase the inefficiency of the control, without bringing any advantages. Two ways to deal with this issue have been proposed: putting a limit on the depth of the look-ahead tree [6] and restricting the class of DESs which can be controlled (i.e., only FSM-implemented DESs can be considered) [10, 11].

2.4.1 Abstracted online control

The abstracted approach for online control uses a limited look-ahead window to achieve higher efficiency and to solve the problems of offline control. The DES system is treated as a black box, which simply outputs events as they happen. These events are matched against the language $L(G)$ which can be generated by the system. The legal language is also available. The control decisions are fed back to the system after an event is intercepted. The controller does not have unlimited exploration

capabilities for the look-ahead window. A number N is chosen, which sets the maximal depth to which the controller can explore the tree structure (Fig. 2.8). Since the depth can be a small number, it is expected that each control decision will be available quickly and the controller will be able to operate with very complex systems by simply examining a small part at a time. Knowledge of the whole system is not required.

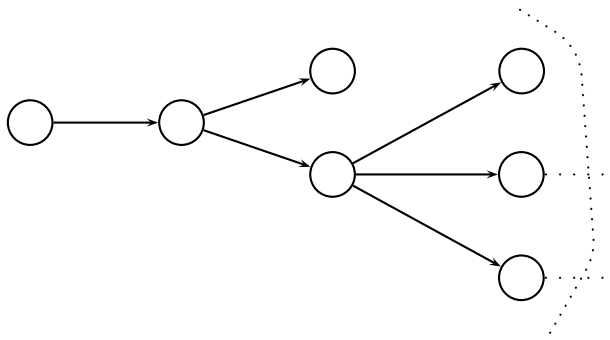


Figure 2.8: Limited look-ahead window (here $N = 3$)

Unfortunately, having a limited view of the whole system does not allow the controller to always make the correct decision. More specifically, the controller does not have any information about what can happen beyond the set depth of exploration. For example, after a controllable event a sequence of uncontrollable events can lead out of the window that is being considered. Up to the border the sequence might be legal, but it is impossible to tell if it is a prefix of a legal sequence or of an illegal sequence (Fig. 2.9(a) and (c), respectively). In general, there are two possible problems: a sequence of uncontrollable events leads out of the limited look-ahead window (as discussed above), or there are controllable events in the sequence, however, if its continuation does not belong to the legal language, late disablement would leave

the system with no way to continue operation (Fig. 2.9(b)). Chung, Lafortune, and Lin propose two policies for how the controller can cope with the uncertainty: the conservative and the optimistic policies [6].

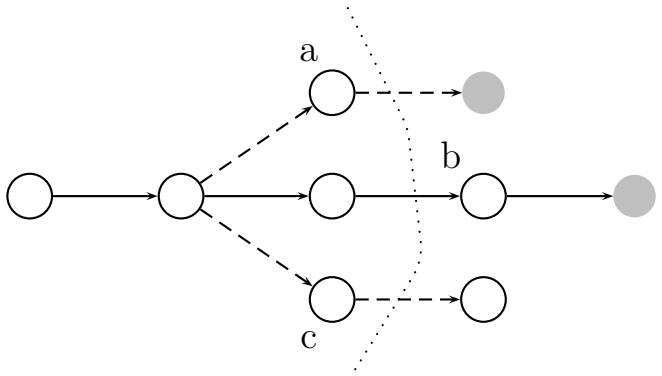


Figure 2.9: Problems with the limited look-ahead window. a) a sequence of uncontrollable events leads to an unwanted state; b) a sequence of controllable events leads to blocking (there are no acceptable continuations); c) a sequence of uncontrollable events leads to a desirable state. Dashed arrows represent uncontrollable events. Dark nodes represent illegal states.

The conservative policy assumes that sequences which cannot be determined to be legal are considered illegal and controllable events leading to them are disabled. This corresponds to the attitude "let's play it safe". This policy guarantees that illegal sequences will never occur. There are two problems with this policy. The first one is that the controller might not produce a correct result, i.e., it will disable legal sequences which otherwise would be available under offline control (Fig. 2.9(c)). The other is the "starting error". A starting error occurs if the controller cannot find a legal sequence in the look-ahead window before the DES starts operating. Thus all controllable events will be disabled even before an event happens. However, if there is no starting error and if the legal language does not contain a sequence

of uncontrollable events with length $N-1$, the conservative-policy controller will exercise control equivalent to the supervisory control for $\sup \underline{C}(K_p, G)$, where K_p is the legal language devoid of sequences containing uncontrollable-event subsequences of length $N-1$ or more. If $L(G, cons)$ denotes the controlled language, then

$$L(G, cons) = \sup \underline{C}(K_p, G),$$

$$\text{where } K_p = K \setminus \{uvw \mid u, w \in \Sigma^*, v \in \Sigma_{uc}^*, |v| = N-1\}.$$

Furthermore, if N is chosen such that it is greater than the maximal number of consecutive uncontrollable events in a sequence from the legal language, then the conservative policy achieves the supremal controllable sublanguage of the legal language, i.e.,

$$N \geq N_u + 2 \Rightarrow L(G, cons) = \sup \underline{C}(K, G),$$

$$\text{where } N_u = \max\{|v| \mid v \in \Sigma_{uc}^*, \exists (u, w \in \Sigma^*) uvw \in K\}.$$

When the optimistic policy is used by a controller, the uncertainties are treated in exactly the opposite way to what the conservative controller would do. If a sequence cannot be determined to be illegal, it is assumed to belong to the legal language. This corresponds to the attitude "let's take a chance". This policy avoids disabling legal sequences unnecessarily, however, it cannot guarantee that illegal sequences will not occur. Like the conservative policy, the result might not be correct, i.e., there is the chance that illegal sequences can happen in the DES (Fig. 2.9(a)). A *runtime error* (RTE) is the condition when the controller disables all controllable events, because the DES is in a state from which no legal sequences can follow. This can happen when the optimistic policy has allowed the occurrence of a prefix of sequences which are all illegal. It is important to note that due to the nature of the optimistic policy,

if an RTE does not occur for any of the sequences executed under the optimistic policy, the controller will achieve the supremal controllable sublanguage of the legal language. So under what condition will RTEs not occur? It is sufficient to choose N greater than N_u, similar to the conservative case. If $L(G, opt)$ denotes the controlled language, then

$$N \geq N_u + 2 \Rightarrow L(G, opt) = \sup \underline{C}(K, G).$$

If there are k modules which constitute a DES, the computational complexity for each control step is $O(k^N|\Sigma|)$. Both policies have the same complexity, since they explore the look-ahead tree in the same fashion. The complexity of the construction of the supremal controllable sublanguage is $O(n^k m|\Sigma|)$, where n is the number of states in each module and m is the number of states in the FSM for the legal language. Based on the "life expectancy" of the DES, one can decide if the offline or the online control will be preferable.

The above results show that the limited look-ahead window approach is viable, especially if N_u can be obtained. Unfortunately, this is not always the case. If the system changes with time, it might be very difficult or even impossible to calculate N_u. There is also another disadvantage: the controller has to perform all computations over and over again for each step of execution. If the DES is in a loop, the controller will not be able to take advantage of the previously obtained results. The second problem is addressed in [10], where online control using state information is discussed.

2.4.2 Online control using state information

Following the work in [6], Hadj-Alouane, Lafortune, and Lin decide to undertake another approach to online control [10]. The controller is given access to the complete

look-ahead tree. However, the controller is limited in another way; it requires access to the underlying FSM implementation and thus it can no longer work with an arbitrary DES (i.e., one not implemented as an FSM). This approach uses states in the DES to store previously computed information about legality and thus avoids repeating the same calculations, one of the problems with the abstracted online control.

First, supremacy-safety of FSM states of G is defined. A state x is said to be *supremacy-safe* if there is a sublanguage of the postlanguage generated by this state, such that the sublanguage is not empty, it is controllable with rescept to the postlanguage of $L(G)$ at this state, and it is a sublanguage of the legal postlanguage of K at this state.

$(\exists S \subseteq L(G)/[x])$

1. $S \neq \emptyset$
2. S controllable with respect to $L(G)/[x]$
3. $S \subseteq K/[x]$,

where $L/[x] = \{s \mid s = tu, t, u \in \Sigma^*, s \in L, \delta(t, q_0) = x\}$ denotes the postlanguage of a language L at the state x. It contains all the sequences from the language L, whose prefixes can be generated by a path from the initial state q_0 to the given state x.

If the system only operates within supremacy-safe states, then the generated language will be a sublanguage of the supremal controllable sublanguage of the legal language. Thus after each event, the controller has exact information about the current state of the DES. Then it explores the look-ahead tree, as deep as necessary, until it can decide for all controllable events if they would lead to states which are not supremacy-safe. The control decision is then based on the results obtained from having unlimited access to the look-ahead tree and the system produces a correct

result, i.e., it achieves the controllable sublanguage of the legal language. It is important to note that since the structure of the FSM is fixed, the supremacy-safety of states does not change over time. This means that if this information is stored, once calculated, the controller will not have to repeat the same calculations. As a result, the controller will start by examining large parts of the look-ahead tree, but as the system evolves, it will have to examine fewer and fewer states, until all required states have been examined. The worst-case complexity for each step is $O(n|\Sigma|)$ and up to $O(n^2|\Sigma|)$ for the whole lifetime of the system, where n is the number of states in the FSM. This result is comparable to the complexity of offline control. However, the approach using state information can perform significantly better if large parts of the FSM lie behind states which are not supremacy-safe (Fig. 2.10).

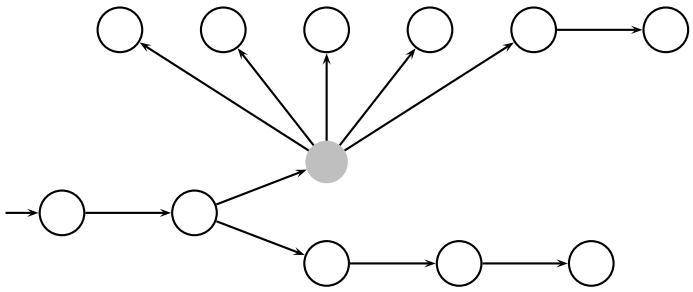


Figure 2.10: A system where large parts are hidden behind a state which is not supremacy-safe. The region beyond the dark state will never be explored.

Of course, a limitation to the depth of the look-ahead tree can also be imposed for this type of online control. Results show that for an identical depth N, this approach produces a better result than the abstracted approach, i.e., the controlled languages for both the conservative and optimistic policies are closer to the supremal controllable sublanguage [10]. Furthermore, a better bound than N_u is determined

for which the controlled language is guaranteed to be equal to $\sup \underline{C}(K, G)$.

As can be seen, the approach using state information offers a solution for one of the issues with the abstracted approach—it is not necessary to repeat calculations—and thus the overall complexity of the algorithm is reduced significantly. Unfortunately, this approach does not help with the other issue—the inability to obtain N_u for DESs which change with time. Furthermore, if it is to be used with such systems, the key advantage of being able to reuse calculations will be lost. Each change of the DES would most probably invalidate the results obtained previously. On the other hand, online control is capable of solving the main issue with offline control: the inability to exercise control due to the state-space complexity. Online control using a limited look-ahead window works by examining only a small part of the whole system at a time. The price for this capability is that the control will not always be optimal or correct.

Further reading on online control can be found in [11], [12], and [20], which deal with the specifics of online control of partially observed DESs, and [9], which presents an algorithm for distributed online control.

Chapter 3

Basis for the work

Two approaches were selected as the basis of my work, since they are already targeted at solving some of the issues in the discussed problem. Modular DES reduces the perceived complexity of large systems and Online Control allows for the control of large systems and systems which vary with time.

3.1 Modular Architecture

As discussed in Section 2.3.2, the modularization of DES systems plays a very important role when large systems have to be modeled and controlled. The definition of highly complex systems can be achieved by modeling subsystems first and then using automated tools to build the complete model using these blocks. Furthermore, modular control can be used in some cases, significantly reducing the complexity of the system supervisor.

There are two characteristics of Modular DESs, which can be exploited conveniently in the control of Dynamic DESs (DDESs). The first one is the hierarchical

structure, implicit in modular systems. Indeed, we need not restrict ourselves with a single level of subsystems. Subsystems of a large system may in turn consist of subsystems, while large systems may be grouped into supersystems. Their designer has a lot of flexibility in terms of the coarseness of detail he or she wishes to model or examine. Tightly related to this property is an even more important characteristic for the DDES. Modularity makes it possible to delimit parts of a complex system which are essentially dynamic. Thus, modules can be used not only to model systems hierarchically, but also to describe the dynamic (time-varying) properties of systems. There is no need to track changes to the complete behavior of a large system if only small parts of it produce variations.

3.2 Online Control

The second body of work on which this work is based is Online Control, as defined in [7] and described in Section 2.4.

Standard supervisory control (SSC) is not suitable at all for the control of DDESs. Each change in the supervised system would invalidate the lengthy computations carried out to build a correct model of the controlled system. The approach used in the online control scheme replaces such demanding computations with series of short calculations which, if the control is exercised over a static system, might end up requiring more processing than the one-time calculation of the SSC. However, when used with DDESs, this "disadvantage" becomes a significant advantage—the system is expected to change frequently enough so that the limited (and adjustable) calculations at each step of the control will outperform the static in nature SSC. There are two flavors of online control: abstracted control and control with state information

(as discussed in Sections 2.4.1 and 2.4.2, respectively). The major difference is that the latter stores some information in the states of the DES, modeled as an FSM, and thus gradually improves its performance (since there is no need to recompute already stored information). Unfortunately, this option cannot be fully utilized if the system is dynamic.

The second major advantage of online control is that it is usable with very large systems, even in cases when the SSC scheme would fail. Since the focus of my work is on the control of large DDES, this property of online control served as an additional motive to choose it as the basis for the proposed control method.

The recent work of Minhas and Wonham [18] is also centered around the idea of modular systems and online control, however, it does not solve the problem of control of dynamic systems and it provides no means for the refinement of control specifications.

Chapter 4

The Dynamic Discrete-Event System Model

In this chapter a formal definition of Dynamic Discrete-Event Systems (DDES) is given. Such systems are constituted of separate small DES modules, which are combined together using synchronous shuffle—i.e., the resulting system can execute all strings that the separate modules can execute, even in interweaved fashion (see Section 2.3.2). The system functions in time; however, time is discrete and it increases by one after each occurrence of an event. A system is called dynamic when the set of modules comprising the DDES can change with time.

Let $M_i = \{M_{1i}, M_{2i}, \ldots, M_{ni}\}$ for $n \in \mathbf{N}$ and some i be a set of DES modules and let $\|M_i$ denote the synchronous shuffle of all elements of M_i. Then DDES $G = \{(\|M_i, i) \mid i \in \{0, 1, \ldots\}\}$. In this sense, i is a time variable and at each specific moment (for each specific value of i) the first element of the pair $(\|M_i, i)$ is a standard DES system. We will use the notation $G_i = \|M_i$.

The above definition defines exactly how systems can vary with time. With each

occurrence of an event, one or more (or all) modules of which the system comprises may disappear and new modules may appear. The modification of a module can be modeled as the replacement of this module with the modified version (i.e., the disappearance of the old module and the appearance of a new module). On the other hand, the probability of a change in the system or the disappearance of specific modules is not defined at all. This is because my work should be applicable to all types of dynamic systems, without regard to the way they evolve. This is especially important in systems, where the properties of the dynamic behavior also change with time (usually such systems would be ones influenced by human behavior, by natural forces, etc.)

The three other characteristics of the type of systems I would like to focus on are not explicitly modeled. The DDES system can be as large as necessary—there is no limit to the size of each constituent module and to the number of modules (except that it is a natural number). However, as already was pointed out, real systems tend to be much larger than a size that is convenient to work with. The continuous life of such systems is implicitly present in the index i, however, my work does not require such behavior. The methods may be applied to finite-time systems and some examples will be provided.

An example of a very simple DDES system can be the store-customer example from Chapter 2. Let us consider a number of customers, each of whom is modeled by a separate module with an index j. For example, for seven customers there are the modules $\{C_j \mid j \in \{1, \ldots, 7\}\}$ and the FSM representation of every module C_j is shown in Fig. 4.1. The variation of the system will be modeled using the sets M_i. If the store has one customer in the beginning, it could be $M_0 = \{C_1\}$. A second customer

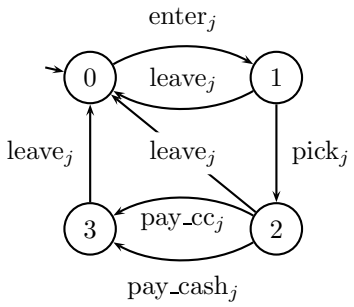


Figure 4.1: DES model of the customer with index j

may appear in time 2 and thus $M_2 = \{C_1, C_2\}$, while M_1 would be the same as M_0. The first customer may execute the events "enter$_1$, pick$_1$". In time 2, the second customer may decide to enter, so "enter$_2$" will happen. Then, the first customer may decide to proceed with "pay_cc$_1$,leave$_1$" and then exit the system. Thus, in time 5, $M_5 = \{C_2\}$. The second customer picks something to buy, "pick$_2$", but when she goes to pay for the item, the store owner recognizes her as an old friend from high school and she ceases to be a customer, so $M_6 = \emptyset$. Thus, the complete DDES G would be modeled as follows: $M_i = \{C_1\}$ for $i \in \{0, 1\}$, $M_i = \{C_1, C_2\}$ for $i \in \{2, 3, 4\}$, $M_i = \{C_2\}$ for $i = 5$ and $M_i = \emptyset$ for $i \in \{6, \ldots\}$, $G = \{(\|M_i, i) \mid i \in \{0, 1, \ldots\}\}$. The system at any given time i would be the synchronous shuffle of all modules in M_i, $G_i = \|M_i$.

Chapter 5

Redundancy for Modular Architecture

The first problem that will be discussed in my work is how to build a system more robust to changes. The controller needs to have knowledge of the system in order to perform its duty. Unfortunately, each time a module in the system changes, this information is invalidated and it is necessary to rebuild the system model. In other words, if the system consists of the synchronous shuffle of modules, this synchronous shuffle would have to be recomputed each time a module appears or disappears. The goal is to choose a method for the rebuilding of the complete system so that the number of calculations that need to be performed is minimal.

A more abstract formulation of the problem would be the following: given elements A_1 to A_n for some $n \in \mathbf{N}$ and a commutative and associative binary operation $\otimes$ that acts on these elements, what is the best method to compute $A = A_1 \otimes A_2 \otimes \ldots \otimes A_n$ so that a change in the number or structure of the elements would result in the least overhead in calculating the new A. The DDES problem we would like to solve may

be cast as this problem by considering the modules as elements and the synchronous shuffle as the binary operation $\otimes$ (since the synchronous shuffle is commutative and associative).

This problem is, to some extent, related to the problem of fault tolerance in system design. Different techniques are applied to try to preserve the correct functionality of a system even when a part of it fails [21, 14]. The basic idea in fault tolerance is redundancy, which may come in many different forms: hardware redundancy, information redundancy, time redundancy, software redundancy, etc. The major difference between fault tolerance and the problem discussed here is that fault tolerance's main goal is to *preserve* the behavior of a system, while our goal is to minimize the computation required to *assimilate* the changing behavior of a system.

Even so, the idea of redundancy still seems very applicable. Keeping in memory redundant byproducts of the computations might significantly decrease the time necessary to rebuild the complete system model when a module changes. I introduce three methods that can be used to speed up this process: stack redundancy, tree redundancy, and hybrid redundancy. Redundancy, although in a very different form and for a different purpose, is also successfully used in simulation [19].

5.1 Stack redundancy

Stack redundancy is achieved by keeping all intermediary steps during the computation of the result A. That is, if we start by computing $A_{12} = A_1 \otimes A_2$, we save this result before we compute the next result, $A_{123} = A_{12} \otimes A_3$. We save separately A_{123} before we proceed and so on. Thus, at the end we have $A_{12}, A_{123}, A_{1234}, \ldots$ to A. I call this method "stack redundancy" because one can imagine that the elements

are stacked on top of each other like a stack (see Fig. 5.1). The calculation of A is performed as $A = (\ldots((A_1 \otimes A_2) \otimes A_3) \otimes \ldots) \otimes A_n$.

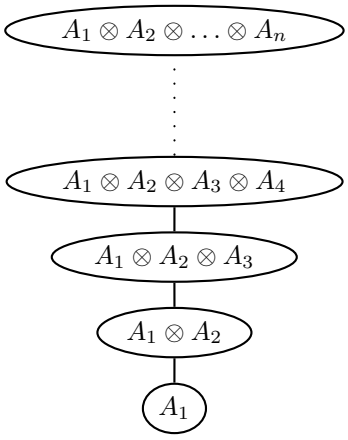
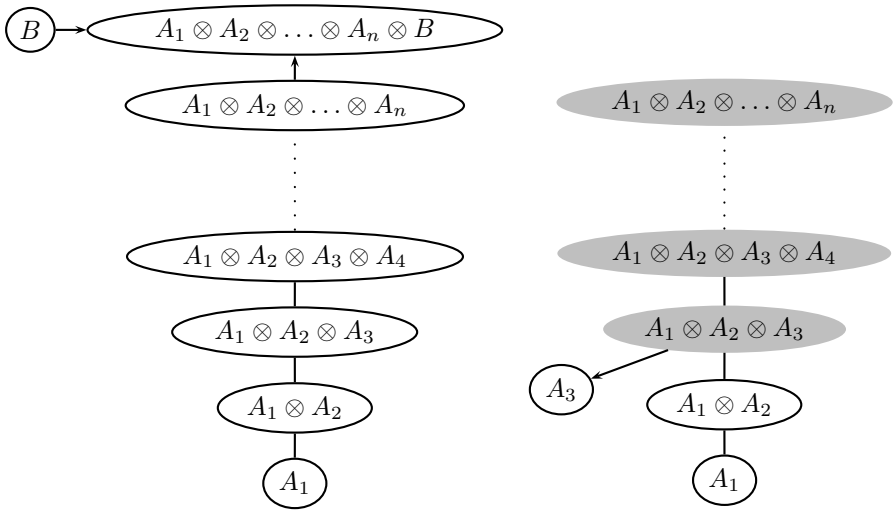


Figure 5.1: Stack-like redundancy structure

Clearly, we are only interested in the final result, A. However, the redundant information we save becomes useful when we need to recompute A if one or more of the constituent elements changes. If element A_i for some $i \in \{1, \ldots, n\}$ changes, then the intermediary result $A_{1\ldots(i-1)}$ is still correct and it is sufficient to perform $n - i + 1$ operations to recompute A. Furthermore, if many elements change, this result still holds true if we take i to be the minimal index of these elements. Correspondingly, if new elements are to be added, the $\otimes$ operation can simply be performed on A. Figure 5.2 illustrates how the redundancy structure can be modified.

The decrease in computational requirements is, however, achieved through higher demands on memory. If the operation $\otimes$ is synchronous shuffle and if the sizes of two elements A_i and A_j are p and q, respectively, then the size of $A_i \otimes A_j$ may be up to $p \times q$. If the size of the largest element is x and there are n elements to be composed, then the total information stored might be up to $\sum_{i=1}^{n} x^i = (1 - x^{n+1})/(1 - x) - 1$,



(a) The addition of the new element B

(b) The removal of the element A_3. All nodes colored dark are affected by the change and have to be reconstructed. The nodes not containing the element being removed are not affected.

Figure 5.2: Diagrams of operations performed on the stack redundancy structure

or up to $2x^n$, as opposed to just x^n if we store only the final result. If x^n is a very large number, then the memory requirements might not be satisfiable.

Stack redundancy is very suitable when a large part of the DDES system is stable, and only a small number of modules vary. An important property of this method is that stable modules will eventually sink to the "bottom of the stack" and it will be necessary to perform just a couple of $\otimes$ operations with the frequently changing modules. In this case, it might be sufficient to keep just a single precomputed result $A_{1\ldots i}$ such that is contains the stable modules.

5.2 Tree redundancy

The next redundancy design I propose is the tree redundancy. Again, the intermediary computational results are kept for future use, however, using a different scheme—one which resembles a binary tree. At each level, $\otimes$ is performed on pairs of the lower-level results. Thus, we start by computing $A_{12} = A_1 \otimes A_2$, $A_{34} = A_3 \otimes A_4$, etc. The next level of the tree will be computed as $A_{1234} = A_{12} \otimes A_{34}$ and so on. This is depicted in Fig. 5.3. As a result, A is computed as $A = (\ldots((A_1 \otimes A_2) \otimes (A_3 \otimes A_4))\ldots) \otimes (\ldots(A_{n-1} \otimes A_n)\ldots)$.

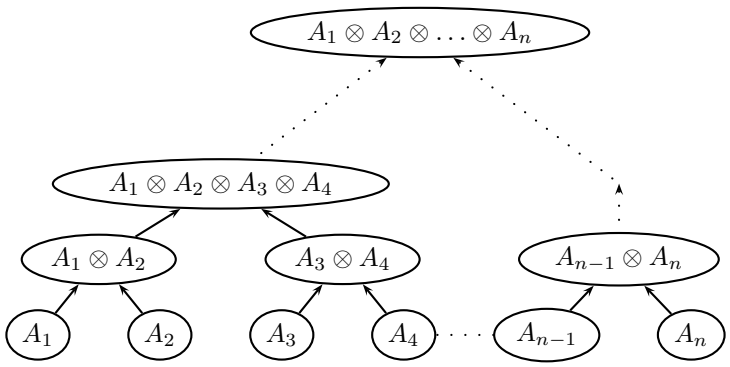


Figure 5.3: Tree-like redundancy structure

What are the advantages that this type of redundancy brings? The most important one is the resistance to random changes in the elements. If the system is highly variable, each module might not be stable for long. In such a case, the stack redundancy would perform very poorly, because as the elements "sink" in the stack, the probability that they will change increases. Thus, the stack will have to be constantly rebuilt and there will be no savings in terms of computations. On the other hand, the distance between every leaf element and the root of the tree is the same, so it

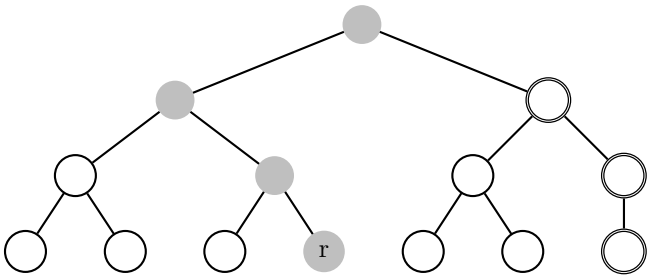
does not matter which element changes. If any element changes, it will require $\log_2 n$ applications of $\otimes$ to rebuild the complete system. If multiple elements in the same subtree change, then they will require less than $\log_2 n$ times the number of changed elements applications of $\otimes$. Furthermore, something similar to the "sinking" of stable elements can be achieved if stable subtrees are preferentially put to the left (or right) of the tree when it is being rebuilt. The algorithm for the rebuilding of the tree structure is shown in Fig. 5.4. Figure 5.5 shows how the algorithm works.

The algorithm for the rebuilding of the tree redundancy structure works very simply. After changes occur in the system, the previous tree structure is partitioned into a set of stable subtrees ST which may contain single elements. This set is fed into the *rebuild* function. It is possible that the previous tree was not fully populated (i.e., it did not contain 2^k elements), so the *chunk* function is invoked to create a set of fully-populated subtrees. It achieves this by recursively splitting incomplete subtrees into three parts: the left subtree of the root, the right subtree of the root, and a single-element subtree consisting of the root. The recursion is invoked only on the right subtree, since the *rebuild* function populates trees from left to right and the left subtrees will always be fully populated.

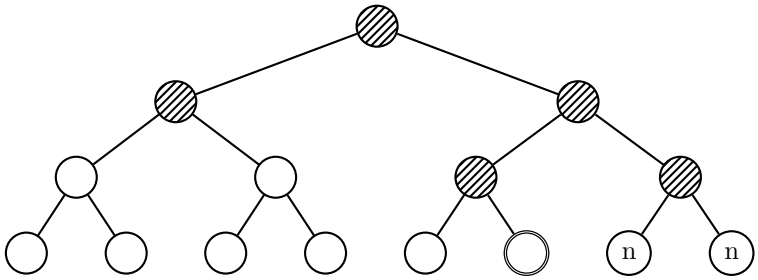
In the next step, the subtrees are sorted first according to height and then, among trees with the same height, according to the time they have been present in the system (i.e., how stable they are). This sorting order allows the placement of stabler subtrees to the left of the rebuilt tree and thus achieves the "sinking" effect discussed previously.

```
1  chunk(T)
2  /* splits a tree T to fully-populated subtrees
3  assumes that all left subtrees of T are fully-populated */
4  if T = null:  return ∅
5  if T.number_of_leaves < 2^T.height
6      R = T.root,  T_l = {R.leftchild},  T_r = {chunk(R.rightchild)}
7      R.leftchild = null,  R.rightchild = null
8      return T_l ∪ T_r ∪ {R}
9  else return {T}
10
11 rebuild(ST)
12 /* ST is a set of all stable subtrees after changes occurred */
13 CT = ∅
14 for each T ∈ ST
15     CT = CT ∪ chunk(T)
16 OT = sort(CT, tree height:descending);
17     sort(same height, time present in system:descending)
18 /* OT is an indexable set
19 level 0 is the leaf level, ⌈log_2 n⌉ is the root level */
20 for i from 1 to ⌈log_2 n⌉
21     NT = ∅,  lastused = false
22     for j from 1 to |OT| − 1
23         if OT[j].height < i
24             T = OT[j].root ⊗ OT[j + 1].root
25             T.leftchild = OT[j],  T.rightchild = OT[j + 1]
26             NT = NT ∪ T
27             j = j + 1
28             if j = |OT|:  lastused = true
29         else NT = NT ∪ OT[j]
30     if lastused = false:  NT = NT ∪ OT[|OT|]
31     OT = NT
32 return OT[1]
```

Figure 5.4: Algorithm for the tree redundancy



(a) The removal of element r. The dark nodes are affected. The nodes with a double border will also be separated in order to create fully-populated subtrees for the tree rebuilding.



(b) The rebuilding of the tree with the addition of two new elements (n). The hatched nodes have to be computed.

Figure 5.5: Diagrams of operations performed on the tree redundancy structure: the removal of an element and the addition of two new elements

In the final stage, the algorithm proceeds by scanning all subtrees for each level from 1 to the root level of the reconstructed tree. At each scan, if trees are found to be of insufficient height, every two neighbors are combined into trees of a higher height. Thus, the complete tree is rebuilt.

Surprisingly, if the operation $\otimes$ is synchronous shuffle, the space occupied by the tree redundancy is smaller than the space required by the stack redundancy, even though there are more "nodes" in the structure. To prove this, let the integer $x > 2$ be the size of the biggest module and let n be the number of modules. We will denote the size of the tree by T_n and the size of the stack by $S_n = \sum_{i=1}^{n} x^i$. The following observation can be made: $T_1 = S_1$, $T_2 = S_2 + x$ and $T_3 = S_3 + 2x$. We will prove that $T_n < S_n$ for any $n > 3$ using induction. First, let us prove that $T_4 < S_4$. The tree will have four leaf nodes, of size x, two inner nodes, of size x^2, and a root node, of size x^4. Thus, $T_4 = 4x + 2x^2 + x^4$ and $S_4 = x + x^2 + x^3 + x^4$.

$$\begin{aligned} 4x + 2x^2 + x^4 &\overset{?}{<} x + x^2 + x^3 + x^4 \\ 3x + x^2 &\overset{?}{<} x^3 \qquad (x \text{ is positive}) \\ 3 + x &\overset{?}{<} x^2 \\ 0 &\overset{?}{<} x^2 - x - 3 \end{aligned} \tag{5.1}$$

The positive root of this equation is approximately 2.3. Thus for all integers $x > 2$: $x^2 - x - 3 > 0$ and thus $S_4 > T_4$. It can be proved analogously that for all $x > 2$, $S_5 > T_5$. The induction hypothesis is that if $T_i < S_i$ for all $i \in \{5, \ldots, k\}$, then $T_{k+1} < S_{k+1}$. Let us consider the two subtrees, L and R, of the root node in a tree with $k + 1$ elements. Let L have l leaf nodes and R have r leaf nodes. From the construction of the tree it follows that $l \geq r$ and $l \geq 4$ (k is at least 5). The number l is less than $k + 1$, so $T_l < S_l$, according to the induction hypothesis. Furthermore,

$T_{k+1} = T_l + T_r + x^{k+1}$. Conversely, $S_{k+1} = S_k + x^{k+1}$.

$$\begin{aligned} T_l + T_r + x^{k+1} &\overset{?}{<} S_k + x^{k+1} \\ (T_l < S_l &\Rightarrow \text{we can substitute}) \\ S_l + T_r + x^{k+1} &\overset{?}{<} S_k + x^{k+1} \\ x + x^2 + \ldots + x^l + T_r &\overset{?}{<} x + x^2 + \ldots + x^k \\ T_r &\overset{?}{<} x^{l+1} + \ldots + x^k \end{aligned} \tag{5.2}$$

The following holds:

- $r = 1 \Rightarrow T_r = S_r$
- $r = 2 \Rightarrow T_r = S_r + x$
- $r = 3 \Rightarrow T_r = S_r + 2x$
- $r \geq 4 \Rightarrow T_r < S_r$ according to the induction hypothesis

We can conclude that in all cases $T_r < S_r + 3x$. Thus, we can substitute and obtain

$$\begin{aligned} T_r &\overset{?}{<} x^{l+1} + \ldots + x^k \\ S_r + 3x &\overset{?}{<} x^{l+1} + \ldots + x^k \\ 4x + x^2 + \ldots + x^r &\overset{?}{<} x^{l+1} + \ldots + x^k \end{aligned} \tag{5.3}$$

Since $l + r = k + 1$ the sum on the left contains one more summand than the sum on the right (the powers go from 1 to r versus $l + 1$ to $l + r - 1$). Each summand on the right is greater than any summand on the left ($x > 2$). Thus, we can cancel some summands and obtain:

$$\begin{aligned} 4x + x^2 &\overset{?}{<} x^k && (x \text{ is positive}) \\ 4 + x &\overset{?}{<} x^{k-1} && (x^2 < x^{k-1}) \\ 4 + x &\overset{?}{<} x^2 \\ 0 &\overset{?}{<} x^2 - x - 4. \end{aligned} \tag{5.4}$$

The positive root of this equation is less than 3, thus we obtain that $T_{k+1} < S_{k+1}$ for any integer $x > 2$. □

The tree redundancy is much more preferable over the stack redundancy when used for DDES purposes. However, the stack redundancy might be preferable when the size of elements does not grow exponentially after the application of $\otimes$. If the operation $\otimes$ does not increase the size of the result, then the stack redundancy would have a size of n, while the tree redundancy would have a size of $2n - 1$.

5.3 Hybrid redundancy

The third type of redundancy I propose is the hybrid type of redundancy which combines ideas introduced in the previous two types of redundancy structures. The basic structure is a binary tree (much like in the tree redundancy), however, inner nodes use the $\otimes$ operation to combine the roots of the two subtrees, as well as a new element (see Fig. 5.6). The result A is computed as $A = (\ldots((A_1 \otimes A_2 \otimes A_3) \otimes A_7 \otimes (A_4 \otimes A_5 \otimes A_6))\ldots) \otimes A_j \otimes (\ldots(A_{i-1} \otimes A_i \otimes A_{i+1})\ldots)$. The major advantages of this new redundancy structure are the reduced size, compared to the tree redundancy, and the "lifting" of elements from the leaves up the tree structure, which increases the chances that fewer than $\log_2 n$ number of $\otimes$ operations will have to be performed when a change in the system occurs.

If the space occupied by the result of the $\otimes$ operation is linear in terms of its operands, the size of the hybrid redundancy can be as much as two times smaller than the size of the tree redundancy. If there are n elements, there would be $2n - 1$ nodes in the tree redundancy, while only n nodes would be in the hybrid structure. However, when the operation $\otimes$ represents the synchronous shuffle of automata (and

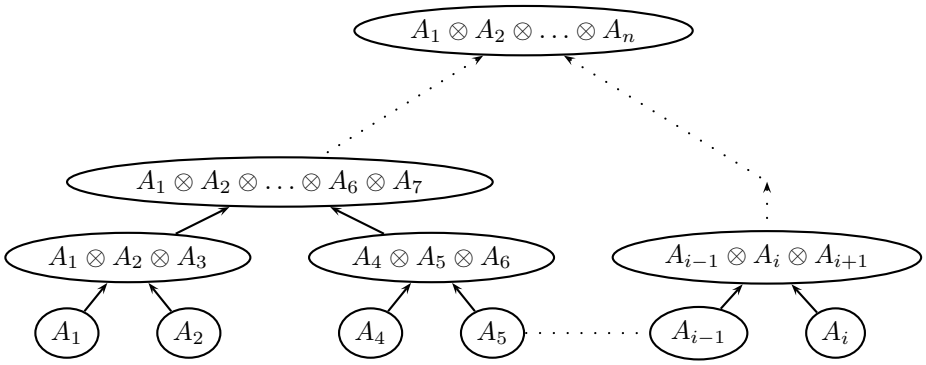


Figure 5.6: Hybrid redundancy structure

hence the size of the result increases exponentially), hybrid redundancy performs almost identically to the tree redundancy. In this case the savings in the size of the structure are insignificant (see Fig. 5.13 for a comparison).

In tree redundancy, it does not make a difference which element changes, since the distance to the root is constant for all elements. With hybrid redundancy, however, the distance between the elements and the root varies, depending on where in the structure the given element resides. Thus, if the changing element is in the level directly beneath the root, only four applications of $\otimes$ would be necessary (two applications per tree level). Thus, elevating frequently-changing elements toward the top of the tree structure will have the same effect as the one observed with stack redundancy. This can be achieved by ordering the elements which will be used to rebuild the tree according to the time they have remained stable and using the stablest elements at the lower levels and the most volatile toward the top of the tree. The algorithm for the rebuilding of the hybrid structure is shown in Fig. 5.7, Fig. 5.8, Fig. 5.9 and Fig. 5.10. Figure 5.11 shows how the algorithm works.

```
1  chunk(T)
2  /* splits a tree T to fully-populated subtrees
3  assumes that all left subtrees of T are fully-populated */
4  if T = null:  return ∅
5  if T.number_of_leaves < 2^T.height
6      R = T.root,  T_l = {R.leftchild},  T_r = {chunk(R.rightchild)}
7      R.leftchild = null,  R.rightchild = null
8      return T_l ∪ T_r ∪ {R}
9  else return {T}
10
11 split(T)
12 /* recursively splits a tree T into all the left subtrees and
13 all the roots of the subtrees leading to the right-most leaf */
14 if T.height = 0:  return {T}
15 C = T.root,  L = C.leftchild,  R = C.rightchild
16 C.leftchild = null,  C.rightchild = null
17 return {C} ∪ {L} ∪ split(R)
18
19 take_last_node(ST)
20 /* removes the right-most leaf of the last subtree in the set
21 ST, splitting the subtree if necessary.  The change to the set
22 is propagated back and the leaf is returned separately */
23 CT = split(ST[|ST|])
24 OT = sort(CT, tree height:descending);
25     sort(same height, time present in system:descending)
26 last = OT[|OT|],  OT = OT − OT[|OT|]
27 ST = (ST − ST[|ST|]) ∪ OT
28 /* the change to ST will propagate to the caller */
29 return last
30
31 height(n)
32 /* returns the height of a tree with n nodes */
33 return ⌊log_2 n⌋
```

Figure 5.7: Algorithm for the hybrid redundancy: support functions (1)

```
1  kappa(n)
2  /* returns the number of nodes in the largest fully-populated
3  tree which can be built using at most n nodes */
4  return 2^κ − 1, such that
5      n = k_1(2^κ − 1) + k_2(2^{κ−1} − 1) + ... + k_{i−1}(2^2 − 1) + k_i
6      is the canonical decomposition of n (k_j ∈ {0, 1, 2})
7
8  leaves(n)
9  /* returns the number of leaves in a tree with n nodes */
10 return ⌈n/2⌉
11
12 compute(r)
13 /* recursively computes the value of the root r of a hybrid
14 structure (i.e., the result A) */
15 if r is already computed or r is a leaf:  return r.value
16 if r.rightchild = null:   r.value = compute(r.leftchild) ⊗ r.element
17 else r.value = compute(r.leftchild) ⊗ r.element ⊗ compute(r.rightchild)
18 return r.value
```

Figure 5.8: Algorithm for the hybrid redundancy: support functions (2)

```
1  rebuild(ST)
2  /* ST is a set of all stable subtrees after changes occurred */
3  f = new(FIFO)
4  n = sum of the number of nodes in all subtrees
5  CT = ∅
6  for each T ∈ ST
7      CT = CT ∪ chunk(T)
8  OT = sort(CT, tree height:descending);
9      sort(same height, time present in system:descending)
10 /* OT is an indexable set */
11 if OT[1].height = height(n):  return OT[1]
```

continued in Fig. 5.10 ...

Figure 5.9: Algorithm for the hybrid redundancy: main function (1)

... continued from Fig. 5.9

```
12  root = take_last_node(OT),  leftnodes = kappa(n−1),  rightnodes = n−1−kappa(n−1)
13  leftsubtrees = {OT[i]  |  i = {1,...,j}, j ≤ |OT|,
14      ∑_1^j OT[i].number_of_leaves = leaves(leftnodes)}
15  rightsubtrees = OT − leftsubtrees
16  f.push(root, leftsubtrees, rightsubtrees, leftnodes, rightnodes)
17  do
18      parent, LS, RS, leftnodes, rightnodes = f.pop
19      if LS[1].height < height(leftnodes)
20          lchild = take_last_node(RS)
21          nextleftnodes = kappa(leftnodes − 1)
22          nextrightnodes = leftnodes − 1 − kappa(leftnodes − 1)
23          leftsubtrees = {LS[i]  |  i = {1,...,j}, j ≤ |LS|,
24              ∑_1^j LS[i].number_of_leaves = leaves(nextleftnodes)}
25          rightsubtrees = LS − leftsubtrees
26          f.push(lchild, leftsubtrees, rightsubtrees, nextleftnodes, nextrightnodes)
27      else lchild = LS[1].root
28      if rightnodes > 0
29          if RS[1].height < height(rightnodes)
30              rchild = take_last_node(RS)
31              nextleftnodes = kappa(rightnodes − 1)
32              nextrightnodes = rightnodes − 1 − kappa(rightnodes − 1)
33              leftsubtrees = {RS[i]  |  i = {1,...,j}, j ≤ |RS|,
34                  ∑_1^j RS[i].number_of_leaves = leaves(nextleftnodes)}
35              rightsubtrees = RS − leftsubtrees
36              f.push(rchild, leftsubtrees, rightsubtrees, nextleftnodes, nextrightnodes)
37          else rchild = null
38      else rchild = null
39      parent.leftchild = lchild,  parent.rightchild = rchild
40  while not f.empty
41  if n = 2^k
42      subroot = root.leftchild
43      leftsubtree = subroot.leftchild,  rightsubtree = subroot.rightchild
44      root.leftchild = leftsubtree,  root.rightchild = subroot
45      subroot.leftchild = rightsubtree,  subroot.rightchild = null
46  compute(root)
```

Figure 5.10: Algorithm for the hybrid redundancy: main function (2)

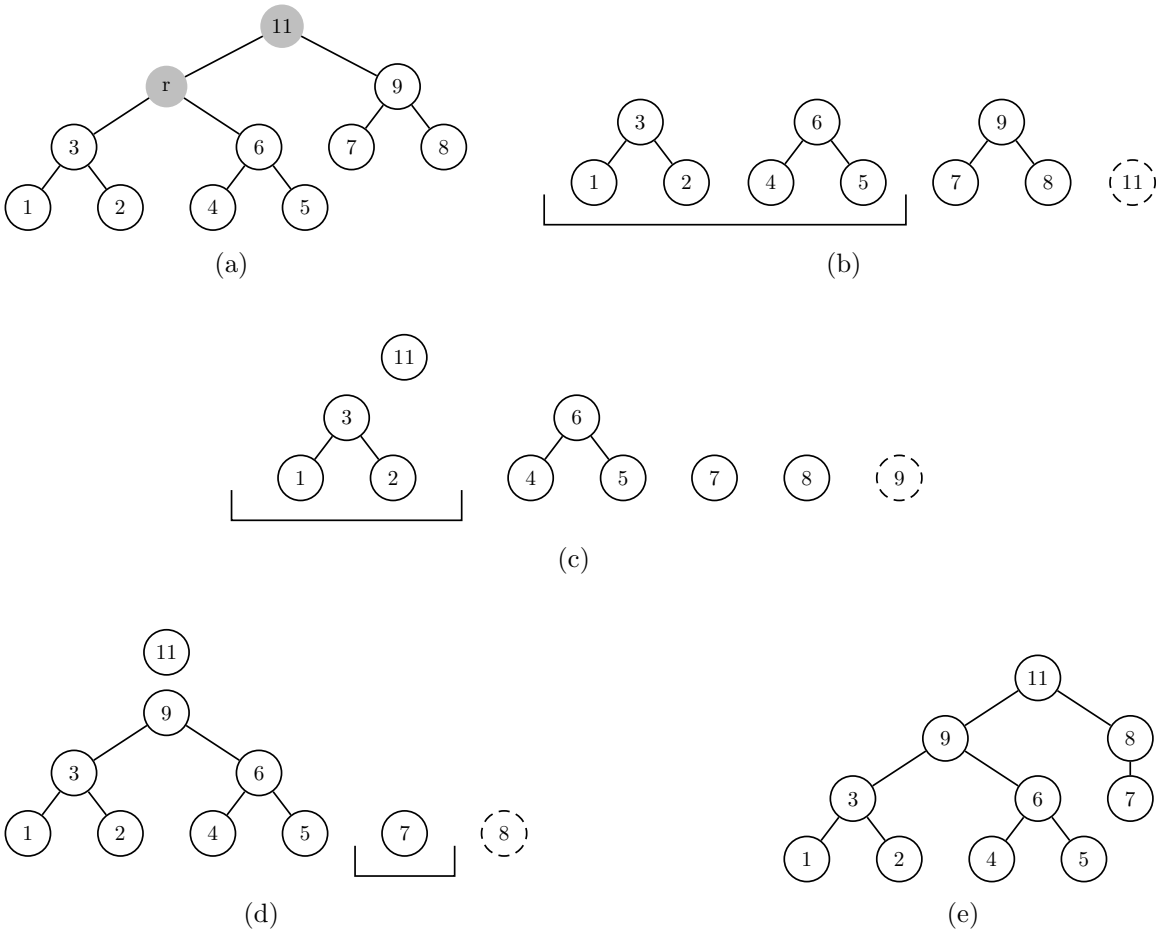


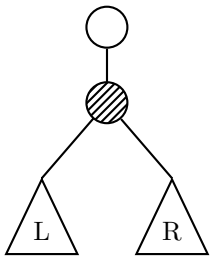
Figure 5.11: Diagrams of the removal of an element and the reconstruction of the hybrid redundancy structure. The node numbers show the relative ordering of the elements according to how much time they are present in the system (lower number means older). (a) The removal of element r. The dark nodes are affected. (b) The first step of the tree rebuilding. The newest element, 11, is chosen to be the root node. Using the remaining nodes, the largest fully-populated subtree that can be built has four leaf nodes. Thus, the first two subtrees are chosen. (c) In the next step, the second newest element, 9, is chosen to be the root node for the subtree. Thus, the third stable subtree (containing elements 7, 8 and 9) has to be split. The left branch of the fully-populated subtree has to have two leaves (half of the total number of leaves for the subtree). (d) The construction of the largest fully-populated subtree is completed. Using the remaining nodes, the largest tree that can be built has one leaf. The third newest element, 8, is chosen for the root of this subtree. (e) The completion of the process of rebuilding. The root node is connected to the subtrees.

The algorithm for the rebuilding of the hybrid redundancy structure starts similarly to the algorithm for the tree redundancy. After changes occur in the system, the previous hybrid structure is partitioned into a set of stable subtrees ST which may contain single elements. This set is fed into the *rebuild* function. It is possible that the previous tree was not fully populated (i.e., it did not contain $2^k - 1$ elements), so the *chunk* function is invoked to create a set of fully-populated subtrees. It achieves this by recursively splitting incomplete subtrees into three parts: the left subtree of the root, the right subtree of the root, and a single-element subtree consisting of the root. The recursion is invoked only on the right subtree, since the *rebuild* function populates trees from left to right and the left subtrees will always be fully populated.

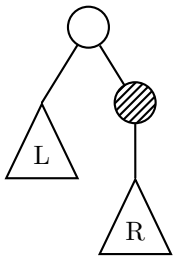
In the next step, the subtrees are sorted first according to height and then, among trees with the same height, according to the time they have been present in the system (i.e., how stable they are). This sorting order allows the placement of stabler subtrees to the left of the rebuilt tree.

The next stage of the algorithm is more complicated since, in order to maximize the benefit of the rebuilt tree, the most frequently-changing (or latest to appear) elements need to be placed toward the top of the tree, while stable structures should not be decomposed if possible. Furthermore, the tallest stable trees should remain on the left, to achieve the "sinking effect". Thus, the algorithm should use a top-down breadth-first method when utilizing frequently-changing elements. This is achieved through the use of a FIFO structure. On the other hand, every subtree should keep track of its own context of lowest-level subtrees, since the algorithm must not use the tallest stable subtrees when building the right branches of the structure. This is achieved by keeping a separate part of the global subtree set ST (or OT,

after sorting) for each subtree to use during the rebuilding. The algorithm uses the *take_last_node* to get the most frequently-changing element available at each time during the process of reconstruction. The variables $leftnodes$ and $rightnodes$ hold the number of nodes which need to be contained in the left or right subtrees, respectively. The sets $leftsubtrees$ and $rightsubtrees$ are used to hold the left and right subtree contexts for each parent node. The only time this algorithm fails to produce a redundancy structure which occupies less space than the tree redundancy is when the number of elements n equals a power of two. This necessitates an increase of the height of the hybrid structure, while the height of the tree structure remains the same. When the number of elements equals a power of two, the hybrid structure looks like the one shown in Fig. 5.12(a). To improve the situation, in the final stage



(a) Unmodified output



(b) The modification performed at the end of the algorithm

Figure 5.12: The output of the hybrid redundancy algorithm when the number of element equals 2^k

the algorithm checks for this condition and then re-arranges the root nodes, as shown in Fig. 5.12(b). This introduces a space saving of $2^{n-1} - 2^{n/2}$. The drawback of this re-arrangement is the chance that the root of a stable subtree could be changed and thus additional computations would be introduced.

Since the evaluation of the root nodes in a top-down approach is not possible (the values of the children are needed first), the *rebuild* function only connects the nodes as it proceeds. Just after the bottom of the tree is reached (i.e., the whole tree is connected), the *compute* function is invoked on the root to perform all necessary calculations and obtain the final result.

Unfortunately, since there are two applications of $\otimes$ per tree level changed, the benefits of the hybrid structure are greatly reduced. If an element changes, this would lead to a maximum of $2\log_2(n/2) = \log_2(n^2/4)$ operations, as opposed to $\log_2 n$ operations with the tree redundancy. Let us count the root as being in level 1, its descendants in level 2 and so on. If a change at level i in the hybrid redundancy is to require fewer computations than a change in the tree redundancy, there should be more than 2^{2i} elements in the structure. For example, a change in the root (level 1) requires two operations $\otimes$. This will be fewer than the number of operations required in a tree redundancy only if there are more than $2^2 = 4$ elements (this would produce three levels in the tree redundancy and a change there would require three operations). We can see that if there are five frequently-changing elements (they would occupy the top three levels in a hybrid structure), the hybrid redundancy will be advantageous only if the total number of elements is more than $2^6 = 64$. Furthermore, this advantage heavily depends on how frequently different elements will change. If there is not a well-defined group of stable elements, the hybrid redundancy will ultimately be computationally more costly.

The applicability of the hybrid redundancy method depends on the nature of the system. In terms of computations, it performs better than the tree redundancy method only when a small number of the elements change and when most of the

system remains stable. For systems where the probability that any of the elements will change is the same, the tree redundancy is clearly preferable. On the other hand, if the system is very large and there is not enough storage space, the hybrid redundancy offers the best performance of all the considered redundancy methods. As Fig. 5.13 shows, however, the decrease of requirements on space is not significant, compared to that of a tree redundancy. Table 5.1 summarizes the properties of the stack, tree, and hybrid redundancies.

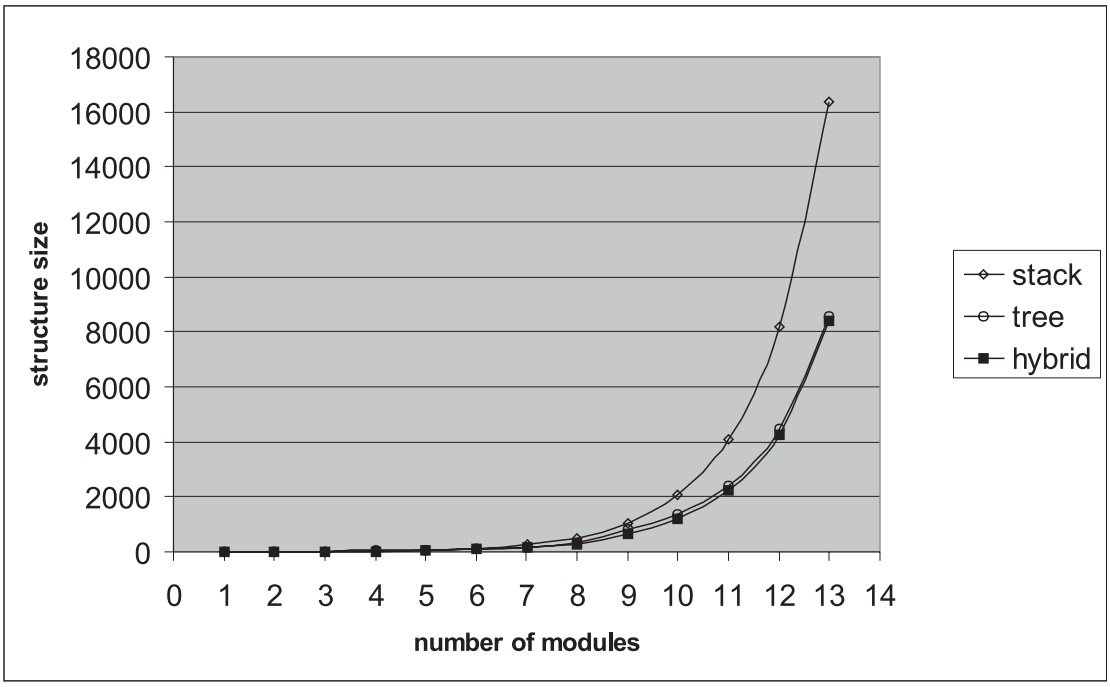


Figure 5.13: Plot of the relation between the number of modules in the system and the size of the different redundancy structures. The size of each separate module is 2.

Table 5.1: Summary of the redundancy structures

	stack	tree	hybrid
size	$\sum_{i=1}^{n} x^i$	$\sum_{i=0}^{\lceil \log_2 n \rceil} (n/2^i) x^{2^i}$	$\sum_{i=1}^{\lceil \log_2 n \rceil} (n/2^i) x^{2^i - 1}$
number of operations if an element changes (on average)	$n/2$	$\lceil \log_2 n \rceil$	$2(\lceil \log_2 n \rceil - 2)$
number of operations if an element changes (worst case)	$n - 1$	$\lceil \log_2 n \rceil$	$2(\lceil \log_2 n \rceil - 1)$
advantages	simple implementation	robust to random changes	small footprint
use when	operation $\otimes$ does not increase the result exponentially	oldest elements have highest chance to change	small storage space

Chapter 6

Control optimization

The next problem that will be discussed is the problem of optimization of the control of DDESs. This problem also relates to the problem of refining the requirements on the behavior of systems. Let us first consider how one would specify the desired behavior of a system. Statements may consist of "the system should do this", "the system should attempt to do this", "the system may do this, but it is undesirable", "the system must not do this", etc. The standard supervision of DESs uses a set of admissible strings as a specification, and the controlled system is not allowed to execute other strings. There are two major drawbacks to this approach: it is not possible to set specifications using all the nuances in the aforementioned statements and the computational complexity for larger systems is daunting. If online supervision is used (see Section 2.4), then in most cases it is even impossible to achieve precisely the behavior specified with the legal set of strings. However, online supervision has a control algorithm that is amenable to modifications. With suitable changes, control of DDES can be significantly improved, resulting both in greater applicability and in a refinement of the control specifications. This is achieved through the use of

a language-based optimization method and the replacement of the scheme used to define specifications.

6.1 Value function

Traditionally, marking (the set of "final" states) is used in finite-state machines to signify "task completion". When a string of events that leads to a marked state is executed, then this string is considered "complete" in some sense. In DESs, marking is used to specify which strings should be executed. However, when used with DDESs, marking does not always make sense and, furthermore, it may lead to problems with the correct interpretation of the behavior of a system. The following issues explain why only prefix-closed languages will be considered in this work.

Let us consider a system which operates continuously, such as a database transaction manager. Which state should be considered marked? If the state where all transactions are either pending or completed is considered as marked, then does it mean that the transaction manager may stop operating at will when such a state occurs? Should the transaction manager start rejecting new transactions because the state when there are no transactions is marked? Obviously, there is more to event strings than just the fact if they lead to a marked state or not. Sometimes it is difficult, if not impossible, to create marking such that it indicates the desired system behavior.

The second problem with marking is related to the specifics of DDESs. The exact behavior of a DDES can be highly unpredictable. Different modules may enter and leave the system and the system will have to adjust. If the DDES is executing a string and a module, needed for the reachability of a marked state, disappears from

the system, would it mean the controller should stop functioning and announce unrecoverable error? Such errors may be preventable in static systems, since the controller can rely on the information about the structure of the systems. With DDESs this is not possible and "unrecoverable" errors may occur very often. How "unrecoverable" such errors actually are cannot be predicted, since the missing module may reappear before its functionality is needed. It is much more natural to expect that the system should try to do the best possible with the resources available.

The third problem arises from the application of online control with a limited look-ahead window. The method has a limited view of how the system can possibly develop. Thus, if the controller steers the system toward one marked state and a critical module disappears, but there is still a way to get to another marked state, the supervisor will not necessarily be able to recognize this option. The problem is illustrated with the system in Fig. 6.1. There is a person wishing to send a parcel with the postal service. The person may bring the parcel to one of two post offices or take it back home ("go_i" and "back_i", respectively). At each office they have an agent who receives the package ("receive_i") and a truck that delivers the parcel to its destination ("deliver_i"). The marked states are the states after a successful delivery. The person may decide to go to the first post office and thus execute "go_1". However, if the system is dynamic, by the time he or she gets to the office, the agent may leave and thus the event "receive_1" cannot happen in the system anymore, so the system cannot get to the marked state after "deliver_1". In such a case the person may return home and decide to go to the second post office, "back_1, go_2". If the system does not change, it would be possible to execute "receive_2, deliver_2" and still reach a marked state. However, a supervisor with a limited look-ahead window may not be able to

recognize this option. If it uses a look-ahead tree of depth 3, after the person goes to the first post office, the supervisor would be able to consider only the sequence "back_1, go_2, receive_2" which does not lead to a marked state. Thus, the supervisor (depending on the decision attitude) would announce a runtime error. However, this would not happen if the language is prefix-closed. In the latter case, the supervisor would enable the "back_1" event and later on be able to consider the successful string "go_2, receive_2, deliver_2".

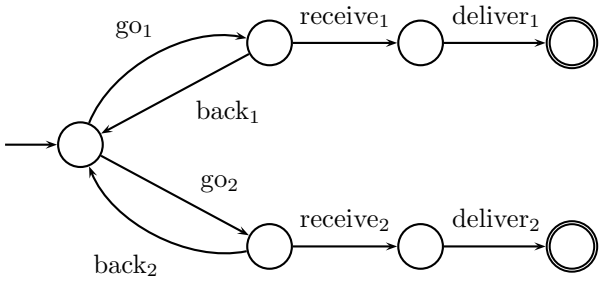


Figure 6.1: DES where a parcel can be delivered using two different post offices

I propose the use of a different and more flexible scheme to define which strings are "complete". Instead of using marking—i.e., a binary operation on the set of states of an FSM—a function $v : L \to \mathbf{R}$, called a *value function*, can be used to define the desired behavior of a DES. The function can be any computable function, which returns how close a string is to a goal the system should achieve. The greater the value of v for a given string, the closer it is to achieving a goal; and conversely, the smaller the value, the farther it is from achieving a goal. It is called a "value function" because the intuition behind it is that it gives the value of a string (which has to be maximized to achieve a goal). Since it returns real numbers, the distance to the goal completion can be fine-tuned to any required level. The supervisors can use

this function to choose the best path to follow and, with a suitable choice of v, errors as described previously can be avoided. Section 6.4.4 explains in more detail how this can be utilized. For practical purposes, the value function should have reasonable restrictions, since its computational complexity affects the overall complexity of the control process.

The second big advantage of the use of a value function is that it can serve as a way to define requirements for the behavior of a system. The simplest function can be one which would return $-\infty$ for all illegal strings (strings outside of the set of legal strings) and 0 in all other cases. As we will see later, this particular function can be used to replace the original control method of online supervision. However, a more elaborate function may be used to refine how "good" or "bad" a string is. Thus, statements of the sort "the system may execute this, but it should be avoided" can be translated into a form usable by the supervisor. For example, a fuzzy logic processor may be used to produce the values of v.

The use of a value function provides many advantages. The disadvantage of the value function is in its increased complexity, compared to simple FSM-based approaches. It presumes that a more powerful computing device will sit at the steering wheel of the system supervision.

6.2 Goals vs. marking

As discussed in the previous section, marking in the system is not suitable for the control of large and dynamic systems and prefix-closed languages are preferable. However, marking signifies the "completion of a task", or what we would like the system to achieve. Thus, in essence, this is a part of the specifications for the behavior

of the system, but it cannot be used if the language is prefix-closed. To solve this problem, instead of using marking in the system I propose that a different scheme be used, namely, the specification of *goals* which will work in conjunction with the value function. A goal is something we would like the system to accomplish and it is an independent entity within the possible behaviors of the DDES. In other words, the event string that achieves the goal has no influence on the future behavior of the system and the events executed after the achievement of the goal do not contribute to the "value" (or benefit) of this goal. Goals are defined by a "goal function" $g : \Sigma^* \rightarrow \{0, 1\}$ which returns 1 if a string signifies the completion of a task and 0 otherwise. The value function v is related to g in the sense that for most $s, t \in \Sigma^*, st \in K, g(s) = 1 : v(st) = v(t)$. In other words, the value function "restarts" calculating after the sting s. The exception to this rule is when the string achieving a goal is the prefix of a string achieving another goal. The function g can be as complex as necessary, however, for most practical purposes it can be interpreted as some kind of marking in the prefix-closed legal language. It is important to make the distinction between real marking and goals. While marking is a hard specification which the system must achieve, goals are used together with the value function and they merely indicate that there is no mutual influence between the value of a string and the values of the continuations of this string.

What is the difference between marking and using goals and the value function instead? First, using a value function allows the definition of "infinite goals", or goals where the length of the event string is infinite (i.e., the supervisor will continuously strive to maximize v for the executed string). The marking scheme is not suitable for this purpose if FSMs are used. Second, the specification of a goal is in terms of

language strings and a function, which allows the supervisor to continue operation even in a frequently-changing environment where the completion of a task may be well beyond the horizon. As discussed previously, if marking is used, this may lead to recurring errors. Third, the value function can be used directly to optimize the control, since the information needed is implicitly available in the function. Marking, on the other hand, has no means to indicate any preference over the way the system operates. Last but not least, it is very hard to predict how marking in the modules will get reflected in the global system, especially if there are many modules. If there have to be changes in the desired outcome of operation, it might be very demanding to figure out how to modify marking in the modules so that the global behavior is changed correctly. Using the goal function g simplifies this task significantly, since the function is already defined globally and furthermore it is independent of the modules of which the DDES consists.

6.3 Optimal control for DDES

There is work which proposes modifications to the control algorithms so that control becomes not only acceptable, but also optimal with respect to some selected criteria [27, 15]. Unfortunately, these approaches are designed only for standard supervisory control, using stable (static) systems. Cost is associated with transitions between states and the algorithms avoid sequences of transitions which will result in high accumulated costs. Such algorithms are not suitable for DDESs, since they do not work with the online control scheme, and since they require access to the FSM representation of the system. In [5] a method for the optimization of online control is presented, however, it is based on a non-standard control framework where the occurrence of

events can be enforced. The value function defined in Section 6.1 does not have these limitations and it can serve as the means to provide optimal control of DDESs. In this section we discuss how this can be achieved through a simple modification of the online supervision algorithm presented in [7]. The role of the value function is similar to the role of the heuristic function for the A* algorithm (described in [1]).

The original supervision algorithm explores recursively the tree of all possible continuations of the currently executed event string. The tree depth is limited to N levels, and thus the controller has a restricted view of the future. Nodes are determined to belong to one of three classes: *legal*, *illegal*, and *undecided*. Legal nodes are nodes that correspond to strings belonging to the supremal controllable sublanguage of the legal language, illegal nodes are nodes that can be determined to correspond to strings leading outside of the supremal controllable sublanguage, and undecided nodes are nodes that cannot be determined to be either legal or illegal. Undecided are nodes at the boundary of the look-ahead tree for which there is insufficient information. Legal nodes are assigned the cost 0, illegal nodes are assigned the cost ∞, and undecided nodes may be assigned 0 or ∞, depending on which control attitude is chosen (see Section 2.4.1). The values are propagated back to the root using a simple approach: if all events leading from a node are controllable, the minimal value from the children is taken (since the supervisor will be able to disable the unwanted events). Otherwise, if there are uncontrollable events leading from the node, the maximal value of the children via the uncontrollable events will be taken (since the controller will not be able to prevent the most costly behavior through uncontrollable events). As well, the supervisor will not explore branches further on if they have infinite costs or if they lead to a marked state with no uncontrollable events leaving that state.

The aforementioned algorithm can be extended very easily to incorporate the information provided by the value function v. Instead of keeping track of which states are legal, illegal or undecided and what control attitude has to be applied, the function v can be simply used to obtain one number which describes all of the above. The smaller the value, the "worse" it is to go through a given path, and $-\infty$ can be regarded as having the same expressive power as the illegality of a state. Since the value function v has a reversed scale compared to the scale of the original online supervision (∞ is used there to denote illegality), the rules for the back-propagation of values have to be dual to the original ones. When a string is valued as $-\infty$ there is no need to search further from this node in the tree. After the achievement of a goal, the system may either stop or continue while attempting to achieve a new goal. Thus, if g of a string equals 1 and there are no uncontrollable events leaving the node, then it is not necessary to explore the look-ahead tree further. The modified algorithm for the control of DDES is presented in Fig. 6.2.

The algorithm I propose is based on the algorithm in [7] and it can be proved that if used in conjunction with a specific value function, v_o, and a specific definition of goals, g_o, then the new algorithm has the same effect as the original version with the optimistic attitude. For a fixed $r \in (-\infty, \infty)$ the function v_o is defined as

$$\begin{aligned} v_o(s) &= -\infty \quad \text{if } s \notin \overline{K}, \\ v_o(s) &= r \qquad \text{otherwise.} \end{aligned} \tag{6.1}$$

The function g_o is defined as $g_o(s) = 1 \Leftrightarrow s \in K$ or, in other words, it selects the strings which are marked in the legal language. To prove that under such conditions the two algorithms are equivalent, let us examine what control decisions are made on all possible inputs.

```
1  cost_to_go(x, h)
2  /* x is a state, h is the string generated to reach this state */
3  Σ_out = events going out of x
4  if x hits the boundary
5      return v(h)
6  elseif g(h) = 1 and Σ_out ∩ Σ_uc = ∅
7      return v(h)
8  elseif v(h) = −∞
9      return −∞
10 else
11     if Σ_out = ∅
12         return v(h)
13     if Σ_out ∩ Σ_uc ≠ ∅
14         for each σ ∈ Σ_out ∩ Σ_uc
15             y = state via σ
16             v_σ = cost_to_go(y, hσ)
17         return min_{σ∈Σ_out∩Σ_uc}{v_σ}
18     else
19         for each σ ∈ Σ_out
20             y = state via σ
21             v_σ = cost_to_go(y, hσ)
22         return max_{σ∈Σ_out}{v_σ}
23
24 control_step(h)
25 /* h is the event string executed so far,
26 E is the set of enabled events for the next step */
27 Σ_out = events going out of root
28 E = Σ_out ∩ Σ_uc
29 if cost_to_go(root, h) = −∞
30     announce RTE
31 else
32 /* y_σ is the state reachable from root via σ */
33     m = max_{σ∈Σ_out}{cost_to_go(y_σ, hσ)}
34     E = E ∪ {σ | σ ∈ Σ_out, cost_to_go(y_σ, hσ) = m}
35 return E
```

Figure 6.2: Optimal DDES control algorithm

For a string s the following notation is defined:

- $|s|$ is the length of s,
- $\Sigma_{out}(s)$ is the set of all events leading out of the state to which s leads,
- $\overline{K}/s|_N = \{t \mid st \in \overline{K}, |t| \leq N\}$ is the set of all suffixes of strings in the legal language K which have s as a prefix and which are of length up to N,
- $X_{mc} = \{t \mid t \in K/s|_{N-1}, \Sigma_{out}(t) \cap \Sigma_{uc} = \emptyset\}$ is the set of suffixes of s in the legal language which lead to marked states with no uncontrollable events leading out. This set is defined for a particular N chosen to be the depth of the look-ahead tree.

Observe that for all strings in X_{mc}, g_o will return 1. The original algorithm considers four major cases with some subcases for the continuations t of the string s:

1. $|t| = N$
   (a) $t \notin \overline{K}/s|_N$
   (b) $t \in \overline{K}/s|_N$
2. $|t| < N \wedge t \in X_{mc}$
3. $|t| < N \wedge t \notin \overline{K}/s|_N$
4. in all other cases
   (a) $\Sigma_{out}(t) \cap \Sigma_{uc} \neq \emptyset$
   (b) $\Sigma_{out}(t) \cap \Sigma_{uc} = \emptyset$

In case (1a), the original algorithm (OA) returns ∞ which stands for illegal/should be avoided. The modified algorithm (MA) for the same input returns $v(st) = -\infty$ which stands for infinite cost/should be avoided. In case (1b) OA returns 0, i.e., this is okay, while MA returns r which evaluates to some payoff or cost, depending on the specific r. However, the controller will only consider it relative to the values of other strings, i.e., it will be equivalent or better (if compared to r or $-\infty$, respectively). In case (2) OA returns 0. Since $t \in X_{mc}$, it means that $g_o(st) = 1$ and MA returns r. In case (3) OA returns ∞, while MA returns $-\infty$. Before we consider the last case, it is important to note that if $t \in \overline{K}/s|_N$ then $st \in \overline{K}$. Furthermore, if $t \notin X_{mc}$ then $g_o(st) = 0$. This allows us to conclude that when OA chooses case (4), the last option block in the MA is executed as well (line 10 of the algorithm in Fig. 6.2). In case (4a) the original algorithm recursively follows the continuations of the string through uncontrollable events and returns the maximal result. Similarly, MA recursively follows the continuations via uncontrollable events, however, it returns the minimal value, since the value function gives the "value" of a string and it is negative values that need to be avoided. In the last case, (4b), OA recursively follows all continuations and returns the minimal result. The modified algorithm, due to its dual nature, performs the same actions, but returns the maximal result. In the original algorithm the control action at the root is such that all controllable events leading to nodes with infinite cost are disabled and the rest of the events are enabled. In the modified algorithm the control action is to enable only the uncontrollable events and the events leading to the nodes with the maximum value. The two control actions are equivalent, since, for any event σ leading from the root, the following holds: $cost\_at\_node(s\sigma) = \infty \Leftrightarrow value\_at\_node(s\sigma) = -\infty$. Furthermore, since both

algorithms return values from a two-element set—$\{0, \infty\}$ and $\{r, -\infty\}$, respectively—the cost of a node will be 0 if and only if the value of the node is r. Lastly, the modified algorithm will enable events leading to all nodes with value r because they all have the same (maximum) value. Formally, for all $\sigma \in \Sigma_{out}(s)$,

- if $\sigma \in \Sigma_{uc}$ then $\sigma \in enabled_{OA}$ and $\sigma \in enabled_{MA}$
- if $\sigma \in \Sigma_c$ and $cost\_at\_node(s\sigma) = \infty$ then $\sigma \notin enabled_{OA}$ and $value\_at\_node(s\sigma) = -\infty$ and $\sigma \notin enabled_{MA}$
- if $\sigma \in \Sigma_c$ and $cost\_at\_node(s\sigma) \neq \infty$ then $\sigma \in enabled_{OA}$ and $value\_at\_node(s\sigma) = r$ and $r = max_{\tau \in \Sigma_{out}(s)}\{value\_at\_node(s\tau)\}$ and $\sigma \in enabled_{MA}$,

where $enabled_A$ stands for the events enabled by the algorithm A after the occurrence of the string s. This concludes the proof of the equivalence between the optimal control algorithm with value function v_o and goals specified by g_o and the original look-ahead algorithm with optimistic attitude. □

It is as simple to modify the value function so that the new algorithm is equivalent to the look-ahead algorithm with conservative attitude.

The implication of the proof above is that the proposed algorithm is at least as powerful as the online control algorithm. The value r was not a priori selected, nor the method used to compute it specified. Thus, if a different value function v is chosen, such that

$$\begin{aligned} v(s) &= -\infty \quad \text{if } s \notin \overline{K}, \\ v(s) &\neq -\infty \quad \text{otherwise,} \end{aligned} \tag{6.2}$$

then this will result in a refinement of the optimal control decisions, i.e., the supervisor may disable some controllable events even if they do not lead to states with a value

of $-\infty$ because they will have a value smaller than the maximum (the valuation of the strings will no longer be constrained to a two-value set $\{-\infty, r\}$). This can be used to advantage by confining the system behavior so that only valuable strings are executed.

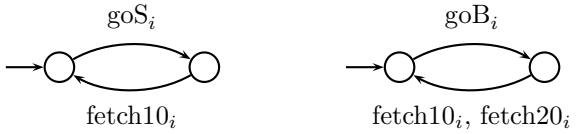
6.4 Issues in optimal DDES control

The control, guided by the value function v, is optimal in the sense that it attempts to steer the DDES so that a goal is achieved as fast as possible and the value of executed strings is highest. All this is possible very trivially, once the required value function is defined. However, this approach does not provide absolute optimal control; it only attempts to provide the best possible approximation by using the function v. There are some optimality issues that arise when this control method is used. The following examples illustrate these issues.

6.4.1 System description

Before the examples are presented, we will provide a brief description of the system which will be explored. Let us consider that we have a company which needs supplies, for example wooden logs. The company uses the logistic services of a provider and the contract is such that we can rent one truck at a time. The provider has two types of trucks available—a small truck (ST) and a big truck (BT)—which can bring ten logs or either ten or twenty logs, respectively. The models are shown in Fig. 6.3(a) and Fig. 6.3(b).

The system consists of the synchronous shuffle of these modules. The number and



(a) small truck ST_i (b) big truck BT_i

Figure 6.3: DES models of trucks

type of trucks available at any given time is not known in advance. Thus the system is inherently dynamic. Let us consider that our company needs a resupply of exactly forty logs and we would like to automate the process, i.e., to have a controller which will guide the system so that we achieve our goal.

The legality specification for the system is given by the following two rules:

1. After one truck goes to fetch supplies, we cannot send another truck (Fig. 6.4).

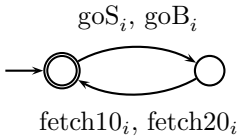


Figure 6.4: First restriction of the legal language, where $i \in \{1, 2, \ldots\}$.

2. The number of logs fetched is forty:

$$\forall s \in K : \#_{\text{fetch10}}(s) \times 10 \ + \ \#_{\text{fetch20}}(s) \times 20 = 40,$$

where $\#_\sigma(s)$ denotes the number of occurrences of σ in s.

The goal function will be defined as $g(s) = 1 \Leftrightarrow \#_{\text{fetch10}}(s) + \#_{\text{fetch20}}(s) \times 2 \geq 4$,

or in other words, a string will achieve the goal if and only if at least forty logs are brought.

The costs associated with the events which can occur in the system are as follows:

- $c(\text{goS}_i) = -100$ (we pay \$100 to rent the small truck)
- $c(\text{goB}_i) = -150$ (we pay \$150 to rent the big truck)
- $c(\text{fetch10}_i) = 500$ (the potential revenue we get from every log is \$50)
- $c(\text{fetch20}_i) = 1000$

and the value function v will be computed using the following equations:

$$\begin{aligned} v(s) &= -\infty && \text{if } s \notin \overline{K}, \\ v(s) &= \textstyle\sum_{s=\sigma_1\sigma_2\ldots\sigma_n} c(\sigma_i) && \text{otherwise,} \end{aligned} \tag{6.3}$$

where $c(\sigma_i)$ is the single-event cost of elements of Σ.

All events in the system are controllable. Even though the proposed algorithm allows for uncontrollable events, the use of controllable events renders the illustrative examples clearer.

6.4.2 Example 1

In this example we will illustrate the effect of an overly limited look-ahead capability. For this purpose, let us limit the prediction of the controller to just one step ahead. At the start the system will have a small truck and a big truck available.

At *time 0* (Fig. 6.5) the system supervisor has to make a decision about which events have to be enabled and which have to be disabled. Since the prediction capability is very limited, the tree is very shallow and simple. The cost of renting a big

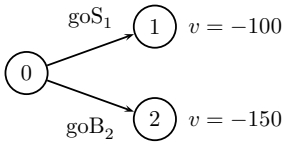


Figure 6.5: Example 1, *time 0* (one small truck, one big truck)

truck is greater than the cost of renting a small truck. Thus the controller, whose task is to optimize the behavior of the system, chooses to disable the event goB_2 and leave only the less costly goS_1.

Since our company needs forty logs, and since sending a small truck four times to bring the logs is more expensive than sending a big truck twice for the same amount of logs, one would correctly observe that it is preferable to send the big truck. This judgment is based on the knowledge of what one expects to happen after a type of truck is dispatched. Unfortunately, the controller is much more limited—it can only foresee one step ahead. Without additional information, the controller will always prefer to send small trucks over big trucks. The fundamental problem illustrated here is that an online supervisor cannot provide optimal control if it cannot observe far enough along event sequences to compute both the relevant costs and the relevant payoffs.

6.4.3 Example 2

In this example we will illustrate the effect of overly lenient look-ahead capability. For this purpose, the supervisor will be able to predict four steps ahead.

The system will be similar to the one in the previous example. At the beginning there will be a small and a big truck available. At *time 2* (after one round-trip) there

will be only a small truck available. At *time 4* (after two round-trips) there will be only a big truck available.

As in the previous example, at *time 0* (Fig. 6.6), the system supervisor has to decide which events should be enabled or disabled. The tree is constructed, only this time it is much deeper and more complex, since the prediction capability is stronger. The branches goS_1goB_2 and goB_2goS_1 are not expanded further, because these sequences are illegal. The supervisor is able to look further into the future and it recognizes that sending the big truck is preferable, since it is cheaper to bring twenty logs at a time (the value function v examines sufficiently long portions of the event sequences). Thus goS_1 is disabled (the payoff through this branch is only 1250, while through the goB_2 branch it is 1700) and the big truck is dispatched.

At *time 1* (Fig. 6.7), the branch where the big truck fetches twenty logs twice is not expanded further, because it achieves a goal and there are no uncontrollable events leading from the state.

At *time 2* (Fig. 6.8(a)), after one round-trip, only the small truck is available (for example, someone else might have rented the big truck). There is only one possible event which also turns out not to be illegal (the value function does not equal minus infinity) and the small truck is dispatched.

After the small truck fetches ten logs, the only available truck is the big truck (for example, the small truck might need maintenance). Thus at *time 4* (Fig. 6.8(c)), the supervisor has to enable goB_2 even though the truck will be used to fetch only ten logs (fetching more than forty logs in total is illegal).

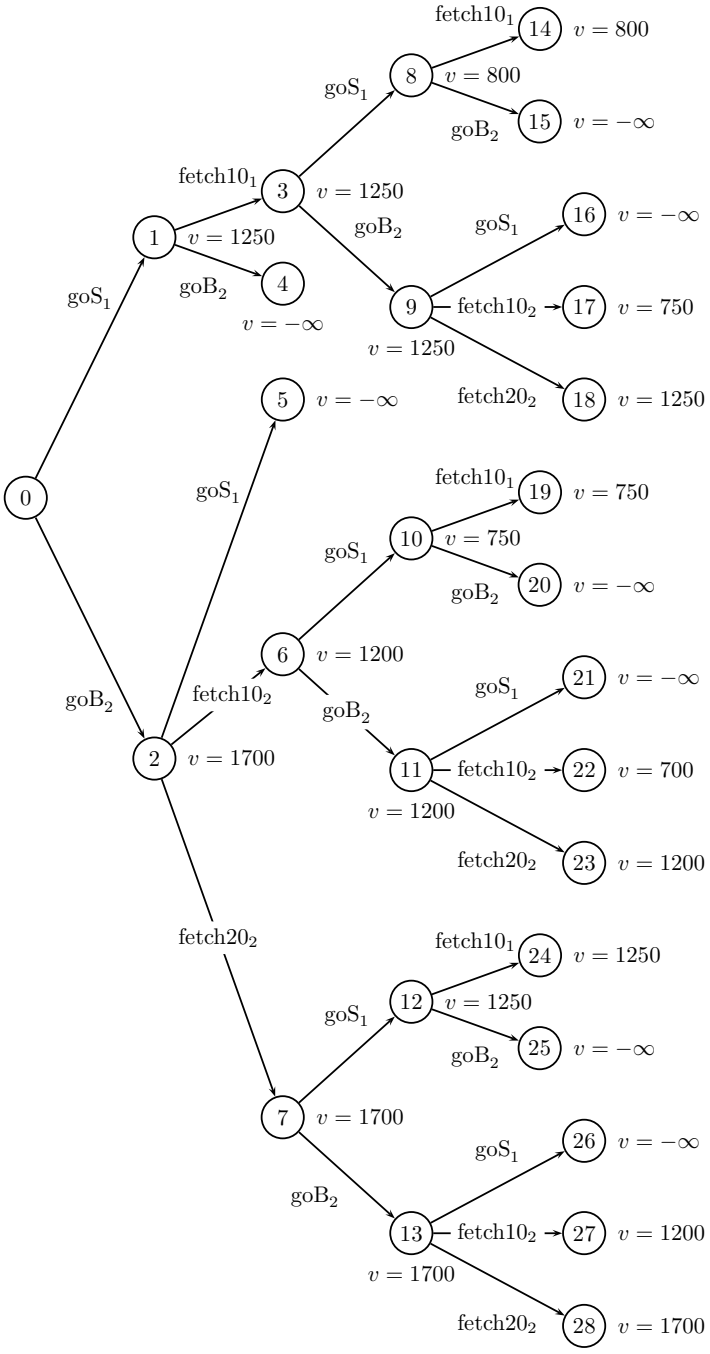


Figure 6.6: Example 2, *time 0* (one small truck, one big truck)

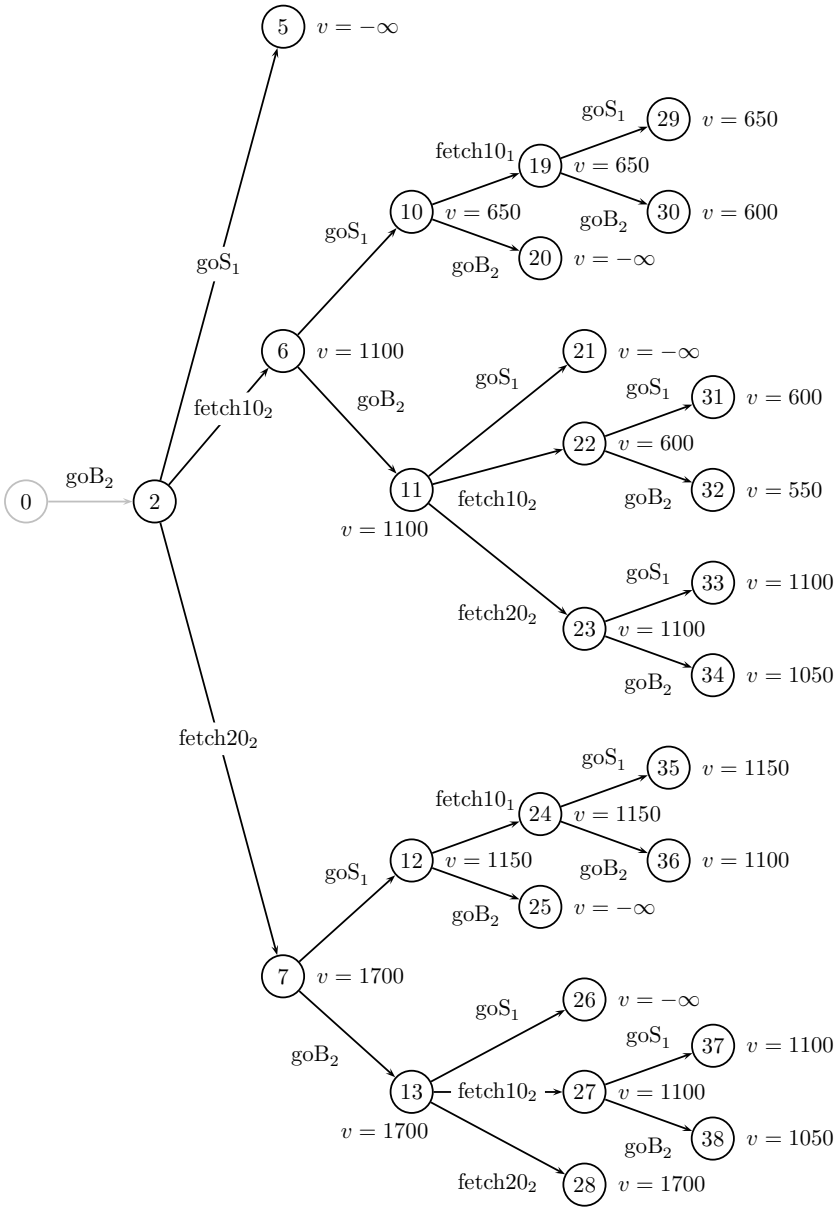
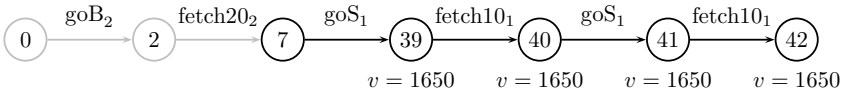
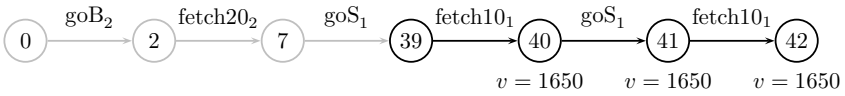


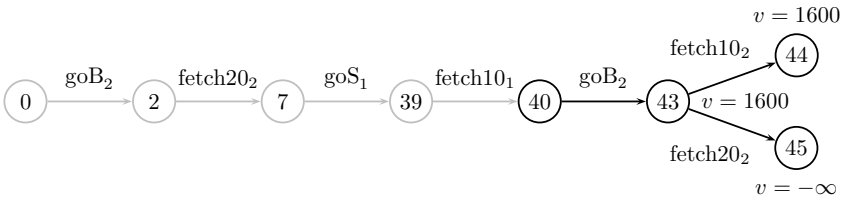
Figure 6.7: Example 2, *time 1* (one small truck, one big truck)



(a) *Time 2* (one small truck)



(b) *Time 3* (one small truck)



(c) *Time 4* (one big truck)

Figure 6.8: Example 2, *time 2* to *4*

After bringing the last ten logs, the goal is accomplished. As one can observe, however, the system ended up incurring a greater cost than necessary. Had we used the small truck at *time 0*, the payoff of fetching forty logs would have been

$$v(\mathrm{goS}_1\mathrm{fetch10}_1\mathrm{goS}_1\mathrm{fetch10}_1\mathrm{goB}_2\mathrm{fetch20}_2) = 1650,$$

while the payoff in this example is

$$v(\mathrm{goB}_2\mathrm{fetch20}_2\mathrm{goS}_1\mathrm{fetch10}_1\mathrm{goB}_2\mathrm{fetch10}_2) = 1600.$$

In other words, the solution produced by the supervisor is not optimal. Unlike the situation in example 1, the problem is not caused by the lack of information: the controller has sufficient look-ahead capability. This time the cause is the availability of incorrect information. Since the system is dynamic, basing control decisions on predictions of a too distant future will most likely result in incorrect assumptions. In this example, at *time 0* the system assumes the big truck will be available in *time 2* and the cost computations are based on this. The fundamental problem illustrated here is that the lack of a good model of the changes of a system renders far-reaching predictions inherently unreliable.

Even though the supervisor cannot always perform perfectly due to the dynamic nature of DDESs, it offers much greater robustness than the original online supervision method [7]. The following example illustrates how the new control algorithm can handle a case where the original approach would produce a runtime error.

6.4.4 Example 3

In this example we will demonstrate how the value function v can be used for more robust control. Let us consider example 2 and let us assume that the big truck can

be used to fetch twenty logs only. Until *time 4* the supervisor will make the same control decisions, since fetching ten logs with the big truck did not have a decisive influence in the previous example.

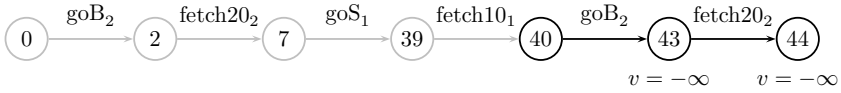


Figure 6.9: Example 3, *time 4*, with legality constraints

At *time 4* (Fig. 6.9), it becomes clear that if the big truck is used, the system will generate an illegal string: fifty instead of forty logs will be fetched. The controller has to disable the goB_2 event and the system gets "stuck" and produces a run-time error. Such errors are naturally expected in online control, but this renders the supervised system unreliable or error prone. One would think that if such a situation occurs, the supervisor should attempt to find the best possible way out automatically.

A way to achieve this "automatic recovery" would be simply to relax the legality requirements and to use the optimizing algorithm to calculate the least damaging course of action. Understandably, sometimes the least damaging course of action may not be admissible at all. In such cases, the hard restrictions will still be kept as legality requirements. For example, if we get a larger resupply of wood logs this will not necessarily result in damage to the company—we might be able to sell the extra merchandise we produce from the logs. On the other hand, not sticking to the rules of the contract and renting more than one truck at a time can result in damage claims or legal actions from the renting company.

Let us examine what would happen if we only keep legal requirement 1 (not more

than one truck can be used at a time) and we modify the value function as follows:

$$
\begin{aligned}
v(s) &= -\infty \quad \text{if } s \notin \overline{K}, \\
v(s) &= \textstyle\sum_{s=\sigma_1\sigma_2\ldots\sigma_n} c(\sigma_i) \\
&\quad \text{if } \#_{\text{fetch10}}(s) + \#_{\text{fetch20}}(s) \times 2 \leq 4, \\
&\quad \text{where } c(\text{fetch10}_i) = 500, \\
&\quad c(\text{fetch20}_i) = 1000, \text{ and} \\
v(s) &= (\textstyle\sum_{s=\sigma_1\sigma_2\ldots\sigma_n} c(\sigma_i)) + 4000 \quad \text{otherwise}, \\
&\quad \text{where } c(\text{fetch10}_i) = -500, \\
&\quad c(\text{fetch20}_i) = -1000.
\end{aligned}
\tag{6.4}
$$

This value function will calculate the first forty logs to be fetched as a benefit (positive value) and all extra logs as costing $50 per log. With this setup the controller can proceed at *time 4* (Fig. 6.10). As one can see, our company still benefits from the

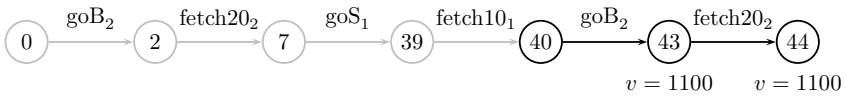


Figure 6.10: Example 3, *time 4*, with value-function constraints

whole operation and the control was optimal, given the circumstances (since it was not possible to predict that the small truck would no longer be available). On the other hand, as before, two trucks are never used at the same time, since this would result in an illegal string.

This example illustrates how the careful delegation of non-hard requirements may result in significant improvement of the overall reliability of the supervised system.

The value function provides the necessary flexibility for constantly changing (dynamic) systems.

6.5 Selecting a limit for the depth of the look-ahead tree

Even though the new control method is capable of dealing better with unexpected situations, we have shown that the exercised control can only be an approximation of the optimal way to guide the system's behavior due to the limited view of the supervisor and the dynamics in the system. The natural question arises, "is there a way to choose a number N to get the best possible performance?" While the best N is always dependent on the specific application, the following observation can be made. We have seen that it is important to have as much information as possible about the contingent future development of the system (i.e., a large N); and to use as little as possible of this information to make the control decisions (i.e., a small N). A possible approach is to put emphasis on the dynamic characteristics of the DDESs.

When the controlled system is highly variable (many changes in its structure occur per some unit time) and it is expected that major modules (i.e., modules very significant for the overall behavior of the system) could appear or disappear, then it is unreasonable to base the control decisions on far-reaching predictions of the executed strings. It is preferable to limit the tree depth to the bare minimum. If the supervisor is to optimize the control decisions, it should have information to be able to compare strings in terms of their value. When comparison of continuations of strings far in the future is not possible, the smallest "local discriminator" should be taken. A local

discriminator is a sequence of events which completes a very limited and simple task and where the immediate cost and payoff of the events is relatively self-contained in the sequence. In the previous examples, the sending of a truck and fetching of logs can be viewed as a local discriminator, since this is a completion of a subtask and the cost and payoff of the two events are mutually related. As we saw, it is not sufficient to have a tree depth of length $N = 1$, most notably because it showed only the initial costs of sending trucks without considering the payoff a new supply would bring. Furthermore, since it is expected that a truck will not "disappear" while on the road, the important modules will be stable for at least two time periods. Thus, if $N = 2$ is chosen, the supervisor would be able to make some reasonable predictions to control the system.

6.6 Threshold for the acceptance of event strings

As we saw in the previous discussion, optimal control is defined as choosing the visible goal with greatest value or the path leading to the optimal goal. This uncertainty is due to the specific value function v used. It is possible that with a limited look-ahead tree a visible goal will be valued more than the limited prefix of an event string leading to the goal having greatest value overall. Furthermore, if it is not possible to choose a limit N according to the guidelines in Section 6.5 (it is difficult to calculate such a number or the number is very large), then one can also imagine a situation when the value of a prefix of a goal is larger than the value of the prefix of another goal, yet when the complete strings are considered, the second goal has greater value than the first one. Even though this problem is inherently not solvable (especially when changes of the system in time are considered), the user of the system might

wish to have more control over how the supervisor treats the values of string prefixes. A tool that might be used for this purpose is a flexible threshold for the enablement or disablement of events. A number τ can be used to specify that if the difference between the value of a string and the maximal value of all strings is less than τ, the string can be considered a valid continuation and the supervisor should not disable it. In this way, the supervisor may be much more lenient and options in the system's behavior would be explored more fully. By increasing τ, the supervisor would be less restrictive, while decreasing τ will result in behavior closer to the optimal. Also, since the number τ is finite, the supervisor would never enable a string with the value $-\infty$ which would be avoided otherwise. The algorithm for the supervisor is modified very little to achieve the above—it is sufficient to change the way E is constructed in the function *control_step* (see Fig. 6.11).

```
1  control_step(h)
2  /* h is the event string executed so far,
3  E is the set of enabled events for the next step */
4  Σ_out = events going out of root
5  E = Σ_out ∩ Σ_uc
6  if cost_to_go(root, h) = −∞
7      announce RTE
8  else
9  /* y_σ is the state reachable from root via σ */
10     m = max_{σ∈Σ_out} {cost_to_go(y_σ, hσ)}
11     E = E ∪ {σ | σ ∈ Σ_out, cost_to_go(y_σ, hσ) ≥ (m − τ)}
12 return E
```

Figure 6.11: Optimal DDES control algorithm with τ-threshold

6.7 Dynamic event evaluation

The greatest strength of the new method comes from the flexibility of the value function. The simple-form value function can be extended to account for other types of dynamics in a system. In particular, let us consider a system where the cost of different events changes with time. This is a very natural feature of many systems. For example, the utilization of a truck can become more expensive with time, due to the increase of maintenance costs. Consequentially, optimization of the control of such systems cannot be based on static event costs, i.e., the value function as defined previous examples cannot be used. However, if we recall the most general specification, v is a function that assigns a real number to a sequence of events. Thus we can use any computable specification to define the function.

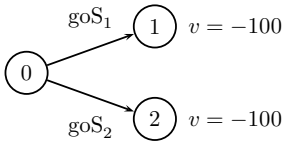
For simplicity, let us use the system defined in Section 6.4 and consider that there are only small trucks available. Let us define the value function as follows:

$$\begin{aligned} v(s) &= -\infty \quad \text{if } s \notin \overline{K}, \\ v(s) &= \textstyle\sum_{s=\sigma_1\sigma_2\ldots\sigma_n,\sigma_i\in\{\text{fetch10}_j\}} c(\sigma_i) - \\ &\quad 100 \times \textstyle\sum_{j=1}^{k} \left(\sum_{l=1}^{ng(j)} \log_2(1+l)\right) \\ &\quad \text{otherwise,} \end{aligned} \tag{6.5}$$

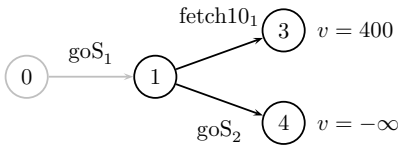
where k is the maximal index of the trucks in the system, $j \in \{1, 2, \ldots, k\}$ and $ng(j) = \#_{\text{goS}_j}(s)$. The only substantial difference from the value function we used in the previous examples is that this one increases logarithmically the cost of using the trucks.

Let us consider a supervisor with a one-step-ahead prediction capability and a system where there are constantly two small trucks available. When the system

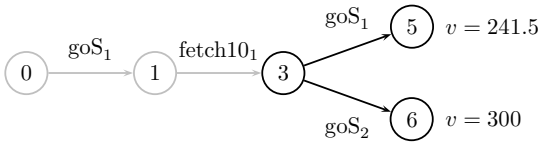
starts (Fig. 6.12(a)), the cost of using either of the two trucks is the same, so the controller leaves both choices available. Let us assume that the first truck is used.



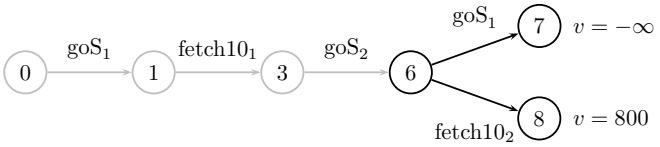
(a) *Time 0*, both events enabled



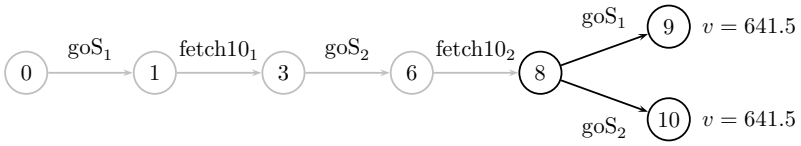
(b) *Time 1*, fetch10$_1$ enabled



(c) *Time 2*, goS$_2$ enabled



(d) *Time 3*, fetch10$_2$ enabled



(e) *Time 4*, both events enabled

Figure 6.12: Look-ahead trees for the system with changing event costs

At *time 2* (Fig. 6.12(c)), using the value function the supervisor correctly computes that sending the second truck instead of the first one will be less costly, since it has not been used at all whereas a truck that has been used requires more expensive

maintenance. This judgment would not have been possible if static event costs were used.

Again at *time 4* (Fig. 6.12(e)), costwise, there is no difference between the two trucks: they have been used an equal number of times. Following this pattern, the value function will attempt to balance the truck usage in the system at any one time. This example illustrates that using the value function, the control of systems may be optimized so that it utilizes resources equally even in systems with complex dynamics.

6.8 Early RTE warning

Finally, we will discuss an interesting and useful characteristic of the optimal control algorithm. Since the supervisor has access to a tree of the possible continuations of the executed event string, it can detect future problems that can occur, such as the system executing a string whose value is $-\infty$, i.e., a string which is not acceptable. In some cases this string contains controllable events and thus the supervisor can disable one of the controllable events and automatically prevent the system from executing the string. In other cases, as discussed in [6, 7], all the events leading to the string may be uncontrollable and the supervisor may not be able to block the execution of the illegal string; and a *runtime error* (RTE) occurs. Runtime errors translate into having a $-\infty$ value at the root of the look-ahead tree. However, often it may be that this value is propagated from branches further in the tree and that the execution of the illegal string is not something imminent (see Fig. 6.13). A large portion of the string can be still acceptable and the system may continue execution. It may be even the case that the illegal string is only one possible continuation and that ultimately it never gets executed: as is the case in Fig. 6.13 if from state 2 the system continues to

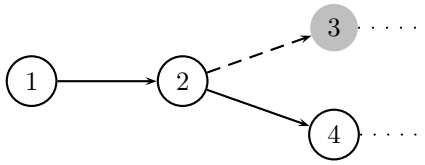


Figure 6.13: Early prediction of a runtime error. State 3 is an illegal state to which an uncontrollable event leads. While in state 1, the controller can predict that from state 2 the system may veer to an undesired path.

state 4. Thus, there is not always a reason to announce RTEs immediately. Instead, an *early RTE warning* can be issued, which would tell the user of the system that there may be trouble down the road. Since the system is dynamic, it may be possible that upon getting such a warning, the user can intervene and modify the system in such a way that the RTE is avoided (for example, an actuator could be added so that an uncontrollable event is rendered controllable). In order to implement this early warning mechanism, the optimal control algorithm should be modified as shown in Fig. 6.14.

After defining the optimal control algorithm and discussing some of its properties, we will see how the different pieces can be brought together to accomplish the overall control of DDESs.

```
1  control_step(h)
2  /* h is the event string executed so far,
3  E is the set of enabled events for the next step */
4  Σ_out = events going out of root
5  E = Σ_out ∩ Σ_uc
6  if ∄σ ∈ Σ_out, v(hσ) ≠ −∞
7      announce RTE
8  else
9  /* y_σ is the state reachable from root via σ */
10     if cost_to_go(root, h) = −∞
11         warn about pending RTE
12         if max_{σ∈Σ_out}{cost_to_go(y_σ, hσ)} = −∞
13             return E ∪ {σ | σ ∈ Σ_out, v(hσ) ≠ −∞}
14     m = max_{σ∈Σ_out}{cost_to_go(y_σ, hσ)}
15     E = E ∪ {σ | σ ∈ Σ_out, cost_to_go(y_σ, hσ) = m}
16 return E
```

Figure 6.14: Optimal DDES control algorithm for early RTE warnings

Chapter 7

Overall DDES control process

In this chapter we describe how the overall process of control of dynamic discrete-event systems would function. The two main ideas behind the proposed method are the persistence of supportive information combined with deferred evaluation. The latter is borrowed from computing algorithms where it has been shown to allow the efficient processing of large amounts of information [13]. The complexity of the DDES control algorithm is also discussed.

7.1 Deferred evaluation

In the previous chapters we saw that, unlike the offline control of DESs (or standard supervisory control), online control requires access to small parts of the DES at a time. However, it was assumed that the model of the DES is readily available. If the system in question is dynamic, this would require that the model is reconstructed each time there is a change. The approaches described in Chapter 5 are designed to alleviate this task, however, they require a lot of storage and may prove not to

be very useful in all cases. Furthermore, if the system is very large, it might be very difficult, if not impossible, to create a complete model of the system even if redundancy structures are not considered. How would an online supervisor be able to operate with such systems?

The solution to this problem is the deferred evaluation, or evaluation-on-demand, of the behavior of the complete system. Instead of calculating the synchronous shuffle of all system modules, and then using the information from the gigantic system to build the look-ahead tree, the supervisor may start building the tree using the information from the separate modules without combining them explicitly. At each state in the tree, the supervisor can scan all the modules and determine which events may happen at this state, using the rule for the computation of the synchronous shuffle. The online control algorithm is forward-searching, with limited depth, and thus it will create and examine only the necessary part of the complete system. A branch in the tree, and thus a part of the synchronous shuffle of all system modules, will be constructed only if there is demand for it, resulting in a deferred evaluation technique.

7.2 State information

In order to support the deferred evaluation method as described in the previous section, it is important that some supportive information is stored in the states of the look-ahead tree.

If the supervisor needs access to all the events which may follow from a state and the model of the complete system is not available, there should be a way to obtain the set of possible continuations. Since the individual modules of the system are accessible, it is sufficient to know which state of every module corresponds to the

node of the tree. Thus, if there are n modules, a simple tuple $q = [q^1, q^2, \ldots, q^n]$ can be associated with each state of the look-ahead tree and q^i would specify the state of module i. The set $\Sigma_{out}(q)$ of all events leading from this state can be constructed as $\Sigma_{out}(q) = \{\sigma \mid \sigma \in \Sigma, \exists i : \delta_i(\sigma, q^i) \text{ is defined}\}$, where δ_i is the transition function of module i. The state $next(q, \sigma)$ which is reachable from $[q^1, q^2, \ldots, q^n]$ via σ can be constructed as follows:

$$\begin{aligned} next(q, \sigma) &= [q_\sigma^1, q_\sigma^2, \ldots, q_\sigma^n], \\ \text{where } q_\sigma^i &= \delta_i(\sigma, q^i) \text{ if } \delta_i(\sigma, q^i) \text{ is defined} \\ q_\sigma^i &= q^i \text{ otherwise.} \end{aligned}$$

If the look-ahead tree is built using the above rules, the result will be equivalent to the look-ahead tree constructed from the synchronous shuffle of all the modules (i.e., the complete system). The proof is trivial.

Once the look-ahead tree is constructed, it can be preserved throughout the process of system control and only needs to be updated at each step. If there is a change in the system, the look-ahead tree has to be rebuilt and this is done automatically by the recursive search of the optimal control algorithm. However, if there is no change, much of the information can be reused. The reusable information includes states determined to have a value of $-\infty$ and states which accomplish goals with determined value. In both these cases, it is not necessary to re-evaluate the continuations beyond these states, since the new evaluation will have the same outcome.

The last piece of information which needs to persist is the history of executed events. The optimal control algorithm uses the value function v which requires this history to compute correctly the value of string continuations. Thus, after an event is executed, the history of events needs to be updated as well.

```
1  DDES_control()
2  h = ε /* initialize the history of events */
3  do
4      if there was system change
5          invalidate look-ahead tree /* will be rebuilt by the
6  control algorithm */
7      enabled_events = control_step(h)
8      if enable_events = ∅
9          return
10     σ =event executed by the system
11     look-ahead tree root=state reachable via σ
12     h = hσ
13     if g(h) = 1
14         h = ε
15 while true
```

Figure 7.1: Main algorithm for the control of DDESs

7.3 Main program

The main algorithm for the control of dynamic discrete-event systems is presented in Fig. 7.1. Purging of the event history upon the achievement of a goal (lines 13 and 14 of the algorithm in Fig. 7.1) is optional. It will speed up subsequent computations, however, if the goal is a part of another goal, this will remove information required by the value function. The choice of whether the event history needs to persist depends on the specific system design.

7.4 Complexity

The overall complexity of the algorithm depends to a great extent on the complexity of the specific value and goal functions, v and g, used. The look-ahead tree exploration algorithm (Fig. 6.2) is a minor modification of the original online control algorithm

[7]; instead of using only 0 and ∞, event strings can be valued using any real number. Thus, the worst-case complexity for the tree exploration is equivalent. It is of the order of $O(k^N|\Sigma|)$, where k is the number of DES modules in the system and N is the depth of the tree. However, this result has to be adjusted to account for the calls to the value and goal functions. In the worst case, both functions may be called at each node of the tree. Thus, the worst-case complexity at each step of operation of the overall algorithm is $O(k^N C(s)|\Sigma|)$, where $C(s)$ stands for the combined complexity of the functions v and g, over an input string s whose length equals the number of executed events since the start of operation. Computations with this complexity have to be performed each time an event occurs and it can be argued that if the lifespan of a system is very long, eventually the accumulated cost would surpass the cost of performing an offline computation of a supervisor for the complete system. However, since dynamic DESs are considered, such an offline computation will be invalidated at each change in the system. Furthermore, the algorithm presented here attempts to guide the system to achieve an event string with maximal value. This is not possible using standard offline supervision.

The next chapter will provide an example of the use of the control method introduced in this work.

Chapter 8

Example and simulation

In this chapter we will present the results of a proof-of-concept simulation which was carried out to compare the true performance of the suggested control method to the performance of the original online control method on which this work is based. For this purpose, the example provided in [7] was used. This example was chosen for two reasons: the system setup is such that it immediately fits in the DDES paradigm, and it offers a very convenient way to compare the algorithms from this work to the original algorithms.

8.1 System description

The system under consideration consists of a number of trains, each modeled as a separate module. Each train can travel along tracks, which are connected in a certain way (see Fig. 8.1). Each track is divided into four sections. All tracks are unidirectional, except track 7 where trains can travel in both directions. There are three stations (S_1 to S_3) and two junctions (J_1 and J_2). Additionally, there is a tunnel

in section 2 of each of track 1 and 4. Trains can enter the system from stations 1 and 2 and can leave the system from stations 2 and 3.

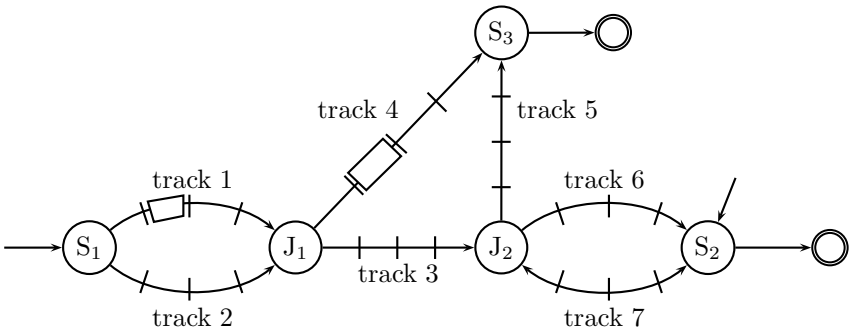


Figure 8.1: Train system

Each train i can execute the following events:

- $t^i_{j,k}$: train i enters section $k \in \{1, \ldots, 4\}$ of track $j \in \{1, \ldots, 6\}$
- $t^i_{j,5}$: train i leaves section 4 of track $j \in \{1, \ldots, 6\}$
- tl^i_k: train i enters section $k \in \{1, \ldots, 4\}$ of track 7 traveling from S_2 to J_2
- tl^i_5: train i leaves section 4 track 7 traveling from S_2 to J_2
- tr^i_k: train i enters section $k \in \{1, \ldots, 4\}$ of track 7 traveling from J_2 to S_2
- tr^i_5: train i leaves section 4 of track 7 traveling from J_2 to S_2
- l^i_j: train i leaves the system permanently from station $j \in \{2, 3\}$

The movement of trains along tracks is controlled by lights, but not all sections have such lights. Thus, the set of controllable events is

$$\Sigma_c = \{t^i_{j,k}, tl^i_k, tr^i_k, l^i_m \mid j \in \{1, \ldots, 6\}, k \in \{1, 3, 5\}, m \in \{2, 3\}\}.$$

The number $i \in \mathbf{N}$ is a natural number, but the number of trains is not a priori known and trains can enter and leave the system at different times, which makes the system a DDES. The set of events in this example does not follow precisely the original example, found in [7]. Namely, the events "train i enters the system" were omitted to prevent the supervisor from having control over when new modules enter the system. This modification does not otherwise have influence on the findings of this simulation.

The legal constraints K considered in the original simulation are:

1. At most one train can occupy any section of the track

   $\forall s \in K, \forall t$ prefix of $s, j \in \{1, \ldots, 6\}, k \in \{1, \ldots, 4\}$:

   $(\#_{t^i_{j,k}}(t) - \#_{t^i_{j,k+1}}(t) \leq 1) \wedge (\#_{tl^i_k}(t) - \#_{tl^i_{k+1}}(t) \leq 1) \wedge$

   $(\#_{tr^i_k}(t) - \#_{tr^i_{k+1}}(t) \leq 1)$

2. At least one section is free between trains on the tracks

   $\forall s \in K, \forall t$ prefix of $s, j \in \{1, \ldots, 6\}, k \in \{1, \ldots, 3\}$:

   $(\#_{t^i_{j,k}}(t) - \#_{t^i_{j,k+1}}(t) = 1 \Rightarrow \#_{t^i_{j,k+1}}(t) - \#_{t^i_{j,k+2}}(t) = 0) \wedge$

   $(\#_{tl^i_k}(t) - \#_{tl^i_{k+1}}(t) = 1 \Rightarrow \#_{tl^i_{k+1}}(t) - \#_{tl^i_{k+2}}(t) = 0) \wedge$

   $(\#_{tr^i_k}(t) - \#_{tr^i_{k+1}}(t) = 1 \Rightarrow \#_{tr^i_{k+1}}(t) - \#_{tr^i_{k+2}}(t) = 0)$

3. At most two trains can occupy every junction

   $\forall s \in K, \forall t$ prefix of $s : (\#_{t^i_{1,5}}(t) + \#_{t^i_{2,5}}(t) - \#_{t^i_{3,1}}(t) - \#_{t^i_{4,1}}(t) \leq 2) \wedge$

   $(\#_{t^i_{3,5}}(t) + \#_{tl^i_5}(t) - \#_{t^i_{5,1}}(t) - \#_{t^i_{6,1}}(t) - \#_{tr^i_1}(t) \leq 2)$

4. All trains traveling simultaneously on track 7 have to travel in the same direction

   $\forall s \in K, \forall t$ prefix of $s : (\#_{tl^i_1}(t) - \#_{tl^i_5}(t) = 0) \vee (\#_{tr^i_1}(t) - \#_{tr^i_5}(t) = 0)$

5. The use of tracks 6 and 7 is balanced with a maximal error of 10

   $\forall s \in K, \forall t$ prefix of $s : abs(\#_{tl_1^i}(t) + \#_{tr_1^i}(t) - \#_{t_{6,1}^i}(t)) \leq 10$

6. The use of the two tunnels is balanced with maximal error of 10

   $\forall s \in K, \forall t$ prefix of $s : abs(\#_{t_{1,3}^i}(t) - \#_{t_{4,3}^i}(t)) \leq 10$

Furthermore, marking in the system is considered: for every train, the stations, the junctions, and the exit points of the system are marked states. In [7] it is determined that a sufficient bound on the depth of the look-ahead tree to ensure correctness of the control for this system and legal language does not exist in general. If we are interested in the prefix closure of the legal language (without the marking), however, choosing N equal to the number of trains in the system would ensure correctness, since the maximal length of a substring of uncontrollable events that can be executed is equal to the number of trains.

8.2 DDES control specifications

In light of the discussion in this work, the following changes were considered when the DDES control method was applied to this example. Instead of marking, goals and the value function were used. A goal was defined as the state when all trains in the system are at a station or the exit states. This does not reflect the marking precisely, but in real life one would not like to have trains in junctions midway to their destinations. Formally, $g(s) = 1 \Leftrightarrow \forall i \in T : last(i, s) \in \{t_{4,5}^i, t_{5,5}^i, t_{6,5}^i, tr_5^i, l_2^i, l_3^i\}$ where T is the set of indexes of all trains in the system and $last(i, s)$ returns the last event for the train i in the string s. The new legal language K' is defined as the prefix closure of the language K without constraint (6). This constraint was omitted because we would

like to show how it can be substituted with a suitable value function. The value function returns $-\infty$ for all strings which do not satisfy the new legality constraints. Furthermore, it is used to attempt to fulfill the following specifications:

1. Trains should move through the system aiming to reach stations, since this is the task in real life. Furthermore, it is not desirable that a train reaches the same station from which it started or that it leaves the system without performing any action. Thus, trains which arrive at a station different from their starting station contribute the value 200 to the overall value of the string, while arrivals at the starting station (this is possible only for station 2) contribute negatively with a value of -200. Similarly, if a train executes l_2^i and station 2 is its starting station, i.e., it leaves the system from station 2 without visiting other stations, the value 200 is subtracted from the overall value of the string.

2. The use of the tracks should be balanced. For every use of a section of a track, a junction or a station (i.e., for every event $t_{j,k}^i$, tl_k^i or tr_k^i), the value of $1 + \ln m$ is subtracted from the overall value returned by the function v, where m stands for the number of times such an event appears in the prefix of the string.

3. Trains should move in an interweaved fashion, i.e., trains should not wait for too long before they move. This is achieved through the use of a priority queue. For each train it is calculated how many time intervals have passed since it has moved last and then all these intervals are summed up and subtracted from the overall value returned by v. Thus, if the train which has waited longest moves, this will minimize the number subtracted (and achieve a better overall value). Since the value function v cannot determine when a train has entered the system simply by examining the event string, it assumes that every train

in the system has waited since the last time it has moved (or from time 0 if it has not moved at all). Thus, trains which enter the system later may have an advantage over trains already in the system. However, we deemed this is not a very serious issue.

4. The use of the two tunnels should be balanced. Instead of using a hard restriction coded in the legality constraints, we decided to demonstrate how the value function can be used for the same purpose. Thus, for every string the difference between the tunnel usage times 10 is subtracted from the overall value of the function v. This method is used to replace legality constraint (6).

All these specifications are reasonable expectations a user of the system would have, however, none of them is strong enough to be defined as a legality constraint (and thus produce a very limited system). Before we define the value function v formally, let us specify the following notation:

- $last(t)$ stands for the last event of the string t,
- $last(t, t^i_{*,1})$ stands for the last event of the string t which belongs to the set $\{t^i_{j,1}, tl^i_1, tr^i_1 \mid i \text{ is an index of a train}, j \in \{1, \ldots, 6\}\}$ and
- $last\_without(s, S) = t$ stands for the longest suffix of the string s such that it does not contain events from the set S, i.e., $s = rt, r \in \Sigma^*, t \in (\Sigma \setminus S)^*$ and $|t| = \max\{|v| \mid s = uv, u \in \Sigma^*, v \in (\Sigma \setminus S)^*\}$

Then, the value function v is defined as follows:

$$
\begin{aligned}
v(s) &= -\infty \quad \text{if } s \notin K', \\
v(s) &= 200\times \\
&\quad \Big|\{t \mid last(t) \in \{t^i_{4,5}, t^i_{5,5}, t^i_{6,5}, tr^i_5\} \text{ for some } i \wedge last(t, t^i_{*,1}) \in \{t^i_{1,1}, t^i_{2,1}\}\}\Big| + \\
&\quad 200 \times \Big|\{t \mid last(t) = t^i_{5,5} \text{ for some } i \wedge last(t, t^i_{*,1}) = tl^i_1\}\Big| - \\
&\quad 200 \times \Big|\{t \mid last(t) \in \{t^i_{6,5}, tr^i_5\} \text{ for some } i \wedge last(t, t^i_{*,1}) = tl^i_1\}\Big| - \\
&\quad 200 \times \Big|\{t \mid last(t) = l^i_2 \text{ for some } i \wedge last(t, t^i_{*,1}) = tl^i_1 \text{ or is ndef}\}\Big| - \\
&\quad \textstyle\sum_t \sum_i \sum_{k\in\{1,\ldots,5\}} (2 + \ln \#_{tl^i_k}(t) + \ln \#_{tr^i_k}(t) + \sum_{j\in\{1,\ldots,6\}} (1 + \ln \#_{t^i_{j,k}}(t))) - \\
&\quad \textstyle\sum_i \Big|last\_without(s, \{t^i_{j,k}, tl^i_k, tr^i_k, l^i_2, l^i_3 \mid j \in \{1,\ldots,6\}, k \in \{1,\ldots,5\}\})\Big| - \\
&\quad 10 \times abs(\textstyle\sum_i \#_{t^i_{1,3}}(s) - \sum_i \#_{t^i_{4,3}}(s)) \\
&\quad \text{otherwise,}
\end{aligned}
\tag{8.1}
$$

where all prefixes t of the string s are considered in the sets and the sum.

8.3 Simulation

The simulation consisted of two parts. First, the original online control algorithm was run to collect performance information not collected in [7] but important for the comparison of the two methods. Then, the algorithm introduced in this work was run to collect the same performance information.

Both algorithms were implemented in the Java programming language. Other implementations may offer better performance (for example using the C language) or, conversely, worse performance, as is the case with the LISP implementation in [7]. Thus, the time spent by the algorithm at each control step should be used for

comparative purposes within this simulation only. The computer used was running the Microsoft Windows XP operating system on a VIA C3 766MHz processor. The Sun Java VM was used which ran with the default 64MB memory allocation pool that fit entirely in the machine RAM.

Every simulation started with a train in station 1 and a train in station 2. After each time interval (the execution of an event), a random number was generated from a uniform distribution in the interval 0 to 1. When the value was greater than 0.8, a new train was introduced in the system if there were fewer than ten trains already in it. Otherwise (when the value was smaller than or equal to 0.8), a new train was introduced if there were fewer than two trains in the system. The newly introduced trains entered the system from station 1 or 2 with equal probability. The trains were taken in order from a pool of trains. When a train left the system, it was returned to this pool, so it was possible that a train with the same index could enter the system numerous times. A train was removed from the system immediately after the corresponding l_j^i event was executed.

For both methods the following performance information was collected:

1. Average number of trains during the simulation.
2. The value of the generated strings at the end of each simulation.
3. The number of trains which arrive at a station during each simulation.
4. The maximal difference in the usage of the tunnels during each simulation.
5. The decision time for every control step in milliseconds.
6. The number of tree nodes inspected at every control step.

7. The number of tree nodes whose value was reused (i.e., it has been computed previously) for every control step. Please note that this is different from the re-utilization statistic provided in [7], where the average ratio of the survived portion of the look-ahead window between steps is shown.

For each pair of tree depth and simulation length, a number of simulation runs were executed and the results were averaged for these runs. The only exceptions are the minimal and maximal values which reflect the overall minimal or maximal value for the given set of runs.

8.4 Results

When the original online control method was simulated, we adopted the optimistic attitude for the algorithm, since it is naturally closest to the actions of the optimal control method. Three sets of runs were performed: ten runs with a simulation length of 40 and a tree depth of 10; ten runs with a simulation length of 40 and a tree depth of 20; and five runs for a simulation of length 100 and a tree depth of 10. The information gathered is shown in Table 8.1(a). The results observed are comparable to the results presented in the original paper [7] with the notable exception of the decision time per step, which improved dramatically with the new implementation. At the end of each run, the produced string was evaluated with the function v to serve as a basis for comparison with the control method proposed in this work. It can be seen from the newly collected data that in terms of our "non-hard expectations" (i.e., the specifications encoded in the value function), the produced strings have little value (the average value of the strings is negative). This is due to the randomness in the execution of events—as long as an event is legal, it is considered for execution and

there is no attempt to guide the system along a path, valuable for the user. Naturally, the increase of tree depth does not have a beneficial effect on the performance of the algorithm.

When the newly proposed control method was simulated, six sets of runs were performed: five with a simulation length of 40 and tree depths of 1, 2, 3, 5, and 7, consisting of ten runs each; and one set of five runs with a length 100 and a tree depth 5. The information gathered is shown in Table 8.1(b). Of the ten (40,7) runs, only two completed successfully. In the other eight cases the Java VM ran out of memory before the completion of the simulation. Information only from the two successful runs is included in the table. Overall, the optimal control algorithm outperforms the original online control algorithm. The execution of events is targeted and thus the resulting value of the event strings is much greater than the one obtained using the original algorithm. Furthermore, as expected, the greater the depth of the look-ahead tree, the better is the performance of the algorithm—both in terms of the string value and the average number of arriving trains. The average number of arriving trains is also greater than the average achieved with the original algorithm. The simulations show that moving the requirement for the balanced use of the tunnels from the legality constraints to the value function was successful. Not only was the use of the tunnels very balanced, but it was even more so than when the legality constraint was used in the original control algorithm. On the other hand, the optimal control algorithm requires many more computational resources than the original algorithm, both in terms of time and space. Almost every branch of the look-ahead tree has to be examined fully and thus the decision time and the number of nodes created are considerably greater than the corresponding values for the original control algorithm.

This is an expected consequence of the new method, however, and the values are still good enough for the practical use of the system in this example. Furthermore, as seen, the proposed control algorithm performs well even with very shallow look-ahead trees; this all depends on the specific value function used. An interesting fact to note is that the average number of trains is much higher in the simulations with the new control algorithm. This is because the value function does not give any incentive to trains exiting the system. Thus trains usually just wait once they reach their destination and exit only when a sufficiently long time period passes and their priority in the queue increases.

The implemented algorithm was capable of early detection of runtime errors as discussed in Section 6.8. It was observed that runtime errors occurred only when the tree depth was 1. During the ten (40,1) simulation runs there were seven RTE warnings and four RTEs were announced. Our suggestion that pending runtime errors are not always critical proved to be correct. Three of the times there was a runtime error warning, the system continued execution and the runtime error did not occur at all. In another case, there was one legal event which was executed between the runtime error warning and the occurrence of the runtime error, possibly giving an opportunity to a human controller to intervene in time to prevent illegal behavior in the system.

Even though the complexity of the system in this example is such that it may be almost impossible to provide a solution using standard supervisory control (there may be up to 35^{10} states in the complete model), the system is still not comprehensive enough when compared to very large and complex systems in real life. The

DDES optimal control algorithm performs well in this case, however, its computational demands may render it inapplicable to larger systems. On the other hand, the original online control method may still perform well with larger systems. This leads me to believe that ultimately a combination of the two approaches would offer the best performance. The original algorithm may be used to verify the legality of very long event strings, while the optimal control algorithm may work with a very limited look-ahead window and be used to select the best path from the verified paths.

Table 8.1: Simulation results for the train example

(a) Using the original online control algorithm [7]

Length of run	Tree depth	Trains (avg)	Value of generated string (min)	Value of generated string (avg)	Value of generated string (max)	Arrivals (avg)	Tunnel balance (avg max)	Decision time per step (ms) (min)	Decision time per step (ms) (avg)	Decision time per step (ms) (max)	Inspected nodes per step (avg)	Reused nodes per step (avg)
40	10	4.3	-1095.8	-583.0	90.5	1	1.3	0	132.8	1612	107.1	1.2
40	20	4.4	-1668.2	-725.3	135.4	0.8	1.4	0	500.7	7711	423.9	1.3
100	10	4.9	-2218.4	-1546.9	-750.8	2.4	2.6	0	581.9	39147	316.6	1.4

(b) Using the DDES optimal control algorithm

Length of run	Tree depth	Trains (avg)	Value of generated string (min)	Value of generated string (avg)	Value of generated string (max)	Arrivals (avg)	Tunnel balance (avg max)	Decision time per step (ms) (min)	Decision time per step (ms) (avg)	Decision time per step (ms) (max)	Inspected nodes per step (avg)	Reused nodes per step (avg)
40	1	4.5	$-\infty$	68.8	335.8	0.8	0.6	0	34.6	350	5.9	0
40	2	5.2	-216.5	-39.0	287.6	0.7	0.9	0	44.4	170	25.2	0.7
40	3	6.5	-70.9	79.2	296.9	1.4	1	10	158.1	911	193.3	4.3
40	5	6.4	-69.1	124.9	217.8	1.7	1	30	2502.9	34810	3487.6	102.1
40	7	5.2	210.9	227.0	243.1	2	1	60	11039.1	104721	14893.0	307.5
100	5	8.4	927.7	1183.3	1291.1	7.8	1.2	20	14593.7	85823	13992.5	155.3

Chapter 9

Conclusion

In this work I have presented a method for the control of dynamic discrete-event systems. The definition of dynamic systems and the suggested optimal control algorithms are novel work, aimed at solving some of the underlying issues in the standard supervisory control theory for discrete-event systems. The major goal was to adapt the concepts of this field and make them more applicable to real-world problems.

There are numerous types of real systems which can be modeled discretely and thus different approaches are needed to control different systems. I focused on a solution for the control of a specific class of discrete-event systems. I considered systems which are dynamic, which are relatively large, which continuously attempt to achieve different goals, and for which the users would wish to have requirements with different levels of stringency. Systems which naturally fit this class are, for example, operating systems and dispatching centers.

Two major topics were discussed: redundancy structures for the efficient reconstruction of the complete system models when constituent modules change, and the

use of online control to optimize the supervision of dynamic systems. Three redundancy structures were proposed: stack redundancy, tree redundancy and hybrid redundancy. Their characteristics and their applicability in different scenarios were discussed. The control algorithms were designed to address the specific requirements of the class of systems considered. The online control paradigm was used since it can adapt automatically to the changes that occur in dynamic systems. A value function was used to guide the supervision of systems, allowing for optimal control and for a flexible way to set requirements on the expected system behavior. This function, combined with the definition of goals, was shown to successfully replace marking in automata and to support the work of continuous-life systems using infinite goals. Additionally, deferred evaluation techniques for online control allow the effective supervision of systems much larger than what would be achievable with standard control. The algorithms were tailored to be easily implementable as modular computer software and separate parts of them can be user-replaced as needed. An extensive example was also provided, illustrating the use of this control method and comparing its performance to that of online control without optimality information. It was shown that the proposed modifications offer a significant improvement in the quality of control of dynamic systems.

This work is only a first attempt at solving the problem of the control of dynamic discrete-event systems. Further research is needed to make this method more effective. One area that can be addressed is setting tighter limits on the exploration of the look-ahead tree. Currently, branches with nodes with a value of minus infinity and nodes which achieve a goal with determined value are not explored further. However, there may be other reasonable conditions under which it is not necessary to explore tree

branches further. It is also possible that some combination of the proposed method and the original online control method may help in the decision process when the look-ahead tree is too large for an exhaustive exploration.

During this work, I noticed that two aspects of the method seem to be able to benefit from the employment of a hierarchical architecture and this may be another area that future research can address. The storage of optimality information to guide the supervisor has high memory demands. One solution is to limit the storage and recompute information on demand. In many cases the information pertains to some higher-level processes, rather than the very low-level event strings which the supervisor controls directly. Thus, one can imagine that abstracting the optimality information to a higher level will reduce the requirements on memory space. The use of a hierarchical architecture for the system supervisor may be a solution. Furthermore, the nature of the online control scheme provides some limited error-prediction capability. Since these errors occur due to insufficient information to take the correct control action, they cannot be prevented without additional intervention. If the supervisors are constructed hierarchically, error notifications can be propagated up until they reach a level capable of making the right intervention. The building blocks are separate modules, thus the actions of higher levels might be the removal or addition of a module so that the problem is solved. This would be a novel approach to the control of discrete-event systems, since so far supervisors have been considered very passive and not able to modify the structure of the controlled system.

Bibliography

[1] Avron Barr and Edward A. Feigenbaum. *The Handbook of Artificial Intelligence*, volume I, chapter II. William Kaufmann, Inc., 1981.

[2] Bertil A. Brandin, Robi Malik, and Petra Malik. Incremental verification and synthesis of discrete-event systems guided by counter-examples. To appear in *IEEE Transactions on Control Systems Technology*, 2004.

[3] Julian Brown. *Minds, Machines, and the Multiverse: The Quest for the Quantum Computer*. Simon & Schuster Inc., New York, New York, USA, 2000.

[4] C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Kluwer Academic Publishers, Norwell, Massachusetts, USA, 1999.

[5] Yi-Liang Chen and Feng Lin. An optimal effective controller for discrete event systems. In *Proceedings of the 40th IEEE Conference on Decision and Control*, volume 5, pages 4092–4097, December 2001.

[6] Sheng-Luen Chung, Stéphane Lafortune, and Feng Lin. Limited lookahead policies in supervisory control of discrete event systems. *IEEE Transactions on Automatic Control*, 37(12):1921–1935, 1992.

[7] Sheng-Luen Chung, Stéphane Lafortune, and Feng Lin. Supervisory control using variable lookahead policies. *Discrete Event Dynamic Systems: Theory and Applications*, 4:237–268, 1994.

[8] Max H. de Queiroz and José E. R. Cury. Modular control of composed systems. In *Proceedings of the 2000 American Control Conference*, volume 6, pages 4051–4055, June 2000.

[9] Nejib Ben Hadj-Alouane, Stéphane Lafortune, and Feng Lin. Think globally, communicate, act locally: On-line parallel/distributed supervisory control. In *Proceedings of the 33rd IEEE Conference on Decision and Control*, volume 4, pages 3661–3666, December 1994.

[10] Nejib Ben Hadj-Alouane, Stéphane Lafortune, and Feng Lin. Variable lookahead supervisory control with state information. *IEEE Transactions on Automatic Control*, 39(12):2398–2410, December 1994.

[11] Nejib Ben Hadj-Alouane, Stéphane Lafortune, and Feng Lin. Centralized and distributed algorithms for on-line synthesis of maximal control policies under partial observation. *Discrete Event Dynamic Systems: Theory and Applications*, 6:379–427, 1996.

[12] M. Heymann and F. Lin. On-line control of partially observed discrete event systems. *Discrete Event Dynamic Systems: Theory and Applications*, 4(3):221–236, 1994.

[13] J. Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989.

[14] Barry W. Johnson. *Design and Analysis of Fault-Tolerant Digital Systems*, chapter 3. Addison-Wesley Publishing Company, 1989.

[15] Ratnesh Kumar and Vijay K. Garg. Optimal supervisory control of discrete event dynamical systems. *SIAM Journal of Control and Optimization*, 33:419–439, 1995.

[16] S. C. Lauzon, A. K. L. Ma, J. K. Mills, and B. Benhabib. Application of discrete-event-system theory to flexible manufacturing. *IEEE Control Systems Magazine*, 16(1):41–48, February 1996.

[17] Feng Lin and Hao Ying. Modeling and control of fuzzy discrete event systems. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 32(4):408–415, August 2002.

[18] Rajinderjeet Minhas and W. M. Wonham. Online supervision of discrete event systems. In *Proceedings of the 2003 American Control Conference*, volume 2, pages 1685–1690, June 2003.

[19] Richard E. Nance, C. Michael Overstreet, and Ernest H. Page. Redundancy in model specifications for discrete event simulation. *ACM Transactions on Modeling and Computer Simulation*, 9(3):254–281, July 1999.

[20] Joseph H. Posser, Moshe Kam, and Harry G. Kwatny. Online supervisor synthesis for partially observed discrete-event systems. *IEEE Transactions on Automatic Control*, 43(11):1630–1634, November 1998.

[21] Dhiraj K. Pradhan. *Fault-Tolerant Computer System Design*, chapter 3. Prentice Hall PTR, Upper Saddle River, New Jersey, USA, 1996.

[22] Ashvin Radiya and Robert G. Sargent. A logic-based foundation of discrete event modeling and simulation. *ACM Transactions on Modeling and Computer Simulation*, 4(1):3–51, January 1994.

[23] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM Journal of Control and Optimization*, 25(1):206–230, 1987.

[24] P. J. Ramadge and W. M. Wonham. The control of discrete event systems. In *Proceedings of the IEEE*, volume 77, pages 81–98, January 1989.

[25] S. L. Ricker and K. Rudie. Know means no: Incorporating knowledge into discrete-event control systems. *IEEE Transactions on Automatic Control*, 45(9):1656–1668, September 2000.

[26] K. Rudie and W. M. Wonham. Think globally, act locally: decentralized supervisory control. *IEEE Transactions on Automatic Control*, 37(11):1692–1708, 1992.

[27] Raja Sengupta and Stéphane Lafortune. An optimal control theory for discrete event systems. *SIAM Journal of Control and Optimization*, 36(2):488–541, March 1998.

[28] Kai C. Wong and W. Murray Wonham. Modular control and coordination of discrete-event systems. *Discrete Event Dynamic Systems: Theory and Applications*, 8:247–297, 1998.

[29] W. M. Wonham and P. J. Ramadge. On the supremal controllable sublanguage of a given language. *SIAM Journal of Control and Optimization*, 25(3):637–659, 1987.

[30] W. M. Wonham and P. J. Ramadge. Modular supervisory control of discrete-event systems. *Mathematics of Control, Signals, and Systems*, 1:13–30, 1988.

[31] T.-S. Yoo and S. Lafortune. A general architecture for decentralized supervisory control of discrete-event systems. *Discrete Event Dynamic Systems: Theory and Applications*, 12:335–377, 2002.

[32] H. Zhong and W. M. Wonham. On the consistency of hierarchical supervision in discrete-event systems. *IEEE Transactions on Automatic Control*, 35(10):1125–1134, 1990.

Glossary

C

conservative policy A policy under which all strings in a look-ahead tree which cannot be determined to be legal are considered illegal, p. 27.

controllable events (Σ_c) Events which can be enabled or disabled (prevented from occurring), p. 10.

D

discrete-event system (DES) A system of discrete states where events (changes of state) happen spontaneously and are not tied to a continuous global time, p. 6.

dynamic discrete-event system (DDES) A discrete-event system which can vary with time: consisting of modules which can appear or disappear as time progresses, p. 36.

F

finite-state machine (FSM) A system of states with labeled transitions between them, p. 6.

G

goal An event string which accomplishes some task specified by the user of a discrete-event system, p. 65.

goal function (g) The function used in the control of dynamic discrete-event systems to identify event strings which accomplish a task. It returns 1 if the string accomplishes a task and 0 if it does not accomplish a task, p. 65.

L

legal language (K) A sublanguage of the language generated by a discrete-event system, containing the acceptable strings, p. 9.

look-ahead tree (look-ahead window) A tree structure containing all event paths which can possibly be executed in the future by a discrete-event system, p. 24.

M

marked state (final state) A state belonging to the set of final states in a finite-state machine, p. 6.

module A small discrete-event system which can be assembled together with other such systems to build a larger discrete-event system, p. 15.

O

offline control Control of discrete-event systems as defined in [23], p. 10.

online control Control of discrete-event systems where the executed events are monitored and guidance is applied as the systems evolve [6], p. 23.

optimistic policy A policy under which all strings in a look-ahead tree which cannot be determined to be illegal are considered legal, p. 28.

P

prefix-closed language A language which contains all prefixes of its strings, p. 7.

R

runtime error (RTE) A condition in online control in which the execution of an illegal string in the look-ahead tree cannot be prevented with the disablement of controllable events, p. 28.

S

supervisor (controller) A unit which is responsible for the guidance of the behavior of a discrete-event system. The supervisor may have different forms. In offline control it is in the form of a finite-state machine. In online control it is in the form of an event monitor, p. 11.

supremal controllable sublanguage The largest sublanguage of a given language which is controllable with respect to a given discrete-event system, p. 13.

synchronous shuffle A parallel composition of discrete-event systems where events can happen in an interweaved fashion and are executed synchronously when possible, p. 16.

U

uncontrollable events (Σ_{uc}) Events which cannot be prevented from occurring, p. 10.

V

value function (v) The function used to optimize the control of dynamic discrete-event systems. It assigns a real number to every string according to its "value" for the user, p. 63.

Common symbols

$\otimes$ A generic commutative and associative binary operation, p. 39.

$\#_\sigma(s)$ The number of occurrences of the event σ in the string s, p. 74.

δ The transition function of a finite-state machine, p. 6.

ϵ The string of length zero, p. 7.

g The goal function used in the control of dynamic discrete-event systems to identify event strings which accomplish a task, p. 65.

K A sublanguage of the language generated by a discrete-event system, containing the acceptable strings; the legal language, p. 13.

$\overline{L}$ A language containing all prefixes of the strings in the language L; the prefix-closure of L, p. 7.

$L(G)$ The language generated by the finite-state machine G, p. 7.

$L_m(G)$ The language accepted by the finite-state machine G, p. 7.

N The depth of a look-ahead tree, p. 26.

N_u The length of the longest substring in a language, containing only uncontrollable events, p. 28.

Q The set of states in a finite-state machine, p. 6.

q_0 The initial state in a finite-state machine, p. 6.

Q_f The set of final (marked) states in a finite-state machine, p. 6.

Σ The set of events which can happen in a discrete-event system, p. 6.

Σ_c The subset of the set of events Σ containing all controllable events, p. 10.

$\Sigma_{out}(s)$ The set of events leading out of the state s (or, if s is a string, out of the state to which s leads), p. 70.

Σ_{uc} The subset of the set of events Σ containing all uncontrollable events, p. 10.

v The value function used to optimize the control of dynamic discrete-event systems, p. 63.

Vita

Lenko Grigorov Grigorov

Education

M.Sc. in Computer Science
Queen's University, Kingston, Ontario, Canada
2004

B.Sc. in Computer Science
Masaryk University, Brno, Czech Republic
2001

Experience

Instructor, 2003
Programming in Java for High School Students
Enrichment Studies, Queen's University

Teaching Assistant, 2002–2003
School of Computing, Queen's University

Software Developer, 2002
Sitius Automation Inc., Markham, Ontario

Web Designer and Developer, 1998–2001
Fulgur Battman Ltd., Brno, Czech Republic

Publications

Baha Jabarin, James Wu, Roel Vertegaal, and Lenko Grigorov. Establishing remote conversations through eye contact with physical awareness proxies. *CHI '03 extended abstracts on Human factors in computer systems.* ACM Press, New York, NY, USA, pp. 948–949, 2003.