Modular Synthesis of Discrete Controllers

Petra Malik Robi Malik David Streader Steve Reeves
Department of Computer Science, University of Waikato
Hamilton, New Zealand
{petra,robi,dstr,stever}@cs.waikato.ac.nz

Abstract

This paper presents supervisory control theory in a process-algebraic setting, and proposes a way of synthesising modular supervisors that guarantee nonblocking. The framework used includes the possibility of hiding actions which results in nondeterminism. As modularity crucially depends on the process equivalence used, the paper studies possible equivalences and points out that, in order to be consistent with respect to the nonblocking property and to supervisor synthesis, a conflict-preserving equivalence must be used. It applies the results to synthesise nonblocking modular supervisors for a manufacturing system.

1 Introduction

Many technical devices in use today are controlled by computer software that is *reactive* in nature, i.e., software that needs to maintain a continuous interaction with its environment. Such software is not only used for everyday household equipment, but also for sophisticated transport or manufacturing systems; application areas include automotive and aircraft electronics, chemical processing plants, and medical equipment. Many of these applications are highly safety-critical: an error in the control software may cause a failure of the system, incurring serious financial consequences or even loss of human life.

Reactive systems interact in a much more complex way than traditional software, which runs once to compute its output from a given input. This makes reactive systems very hard to understand and to get right. With the software complexity continuously increasing over the past decades, the need for formal methods and tools to support engineers in the development of fault-free reactive software has become obvious. Building such tools requires sound mathematical models to describe and understand the dynamics of reactive system behaviour.

This paper compares and combines two approaches for modelling reactive systems that come from quite different backgrounds, namely *supervisory control theory* of discrete event systems, developed by control engineers, and *process algebras*, developed within the computer science community.

Consider a small manufacturing system that consists of two devices running in parallel and linked by a two-place buffer. The first device, called handler, fetches a workpiece and then puts it into the buffer, from where the second device can collect it for further processing. A controller to ensure that the buffer does not overflow can simply prevent the handler from putting a workpiece into the buffer when the buffer is full. Now assume further that the controller can only influence the fetching. Once the handler has got a workpiece, it cannot be prevented from placing it into the buffer. What does the controller need to look like in this scenario? Can buffer overflow be prevented more indirectly, by controlling the fetching, for example? This is the sort of problem supervisory control theory can solve.

While it is not too difficult to develop a controller for the small example given above, the task gets quickly involved when considering manufacturing plants with different sorts of devices, buffers, and workpieces. Soon, the controllers get huge and incomprehensible for humans, and for more complex examples even unmanageable for a computer. Research in process algebra has focused on general ways of composing systems, and to abstract or hide aspects of a system that are irrelevant in a particular context. This allows the modelling of and reasoning about very large and complex systems in a modular way. Combined with the features and methodologies of supervisory control theory, this provides a powerful modelling framework for reactive control software.

In the following, section 2 introduces the relevant concepts and notations from supervisory control theory and process algebra. In section 3, supervisory control synthesis is extended to handle the nondeterministic framework considered in this paper. Section 4 explains the idea of modular synthesis using the example of a manufacturing system. Section 5 presents the main results and shows that congruence and thus modularity can be achieved by using a

conflict-preserving equivalence. In section 6, these results are applied to synthesise a modular supervisor for the manufacturing system. Finally, section 7 discusses some related work, and section 8 closes with some concluding remarks.

2 Notation

2.1 Languages

Traces and languages are simple means to describe reactive system behaviours. Their basic building blocks are *actions*, also called *events*, which are taken from a finite *alphabet* $\mathbf{A}$.

There are two special actions: the *hidden* action τ and the *termination* action ω. The hidden action τ is commonly used in process algebras to describe behaviour that is internal to the system being modelled, that is, behaviour that cannot be observed or controlled by other systems or system components. The termination action ω is introduced to describe the concept of blocking from supervisory control theory. The two actions τ and ω do not belong to $\mathbf{A}$, if they are to be included, the alphabets $\mathbf{A}_\tau = \mathbf{A} \cup \{\tau\}$ and $\mathbf{A}_\omega = \mathbf{A} \cup \{\omega\}$ are used instead.

$\mathbf{A}^*$ denotes the set of all finite *strings* or *traces* of the form $\alpha_1\alpha_2\cdots\alpha_k$ of actions from $\mathbf{A}$, including the *empty trace* ε. A *language* over $\mathbf{A}$ is any subset $L \subseteq \mathbf{A}^*$. The *concatenation* of two traces $s, t \in \mathbf{A}^*$ is written as st. The concatenation of languages $L_1, L_2 \subseteq \mathbf{A}^*$ is defined as $L_1L_2 = \{\, st \in \mathbf{A}^* \mid s \in L_1 \text{ and } t \in L_2 \,\}$. The *prefix-closure* $\overline{L}$ of $L \subseteq \mathbf{A}^*$ is the set of all prefixes of traces in L, $\overline{L} = \{\, s \in \mathbf{A}^* \mid st \in L \text{ for some } t \in \mathbf{A}^* \,\}$. A language L is called *prefix-closed* if $L = \overline{L}$.

2.2 Processes

Processes are modelled using nondeterministic *labelled transition systems* $P = (\mathbf{A}, Q, \rightarrow, Q^\circ, Q^\omega)$, where $\mathbf{A}$ is the alphabet of actions, Q is the set of *states*, $\rightarrow \subseteq Q \times \mathbf{A}_\tau \times Q$ is the *transition relation*, $Q^\circ \subseteq Q$ is the set of *initial states*, and $Q^\omega \subseteq Q$ is the set of *terminal states*.

Processes are represented graphically as shown in figure 1: states are represented as nodes, with the initial state highlighted by a thick border and terminal states shaded grey. The transition relation is represented by labelled edges. Process $\mathbf{Handler}_i$ in figure 1, for example, models a simple device that fetches a workpiece and then puts it somewhere else. Process $\mathbf{Buffer}_i$ models a two-place buffer.

Supervisory control theory uses the set Q^ω of terminal states to represent the possibility of successful termination. Process $\mathbf{Handler}_i$ shown in figure 1, for example, may successfully terminate only if it is idle. To translate this into a process-algebraic action representation, every process is

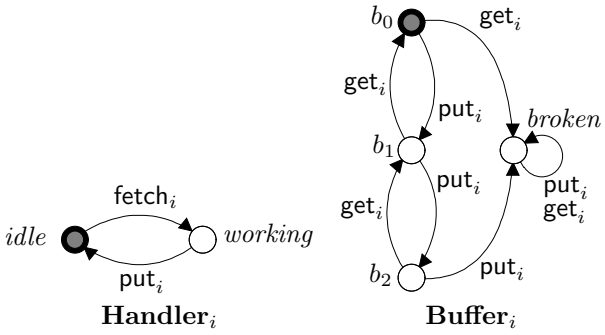


Figure 1. Processes for a small factory system.

assumed to have a terminal state $\perp \in Q \setminus Q^\omega$, from which there are no outgoing transitions. Then the transition relation is extended by adding transitions

$$q^\omega \xrightarrow{\omega} \perp \quad \text{for each} \quad q^\omega \in Q^\omega \,. \tag{1}$$

This construction makes it possible to represent termination only by means of the termination action ω which, if it occurs, is always the last action of a trace.

The action-labelled transition relation $\rightarrow$ is further extended to a transition relation $\Rightarrow \subseteq Q \times \mathbf{A}_\omega^* \times Q$ labelled with traces,

$$q \overset{\varepsilon}{\Rightarrow} q' \quad \text{if} \quad q = q_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} q_n = q' \text{ for some } q_1, \ldots, q_n \in Q; \tag{2}$$

$$q \overset{s\alpha}{\Rightarrow} q' \quad \text{if} \quad q \overset{s}{\Rightarrow} q_s \xrightarrow{\alpha} q_{s\alpha} \overset{\varepsilon}{\Rightarrow} q' \text{ for some } q_s, q_{s\alpha} \in Q. \tag{3}$$

The set of all processes with action alphabet $\mathbf{A}$ is denoted by $\Pi_\mathbf{A}$. The transition relation is also defined for processes, denoting by

$$P \overset{s}{\Rightarrow} P' \tag{4}$$

that process $P \in \Pi_\mathbf{A}$ evolves into $P' \in \Pi_\mathbf{A}$ by executing actions $s \in \mathbf{A}^*$. This is defined as

$$(\mathbf{A}, Q, \rightarrow, Q^\circ, Q^\omega) \overset{s}{\Rightarrow} (\mathbf{A}, Q, \rightarrow, \{q\}, Q^\omega) \tag{5}$$

if $q^\circ \overset{s}{\Rightarrow} q$ for some $q^\circ \in Q^\circ$. The notation $P \overset{s}{\Rightarrow}$ means that $P \overset{s}{\Rightarrow} P'$ for some $P' \in \Pi_\mathbf{A}$.

The possible behaviours of a process are defined by the set of action sequences or *traces* it can execute. The *language* $L(P)$ of $P \in \Pi_\mathbf{A}$ is defined as

$$L(P) \overset{\text{def}}{=} \{\, s \in \mathbf{A}_\omega^* \mid P \overset{s}{\Rightarrow} \,\} \,. \tag{6}$$

$L(P)$ is prefix-closed and contains all complete or incomplete traces that can be executed by a process, including or

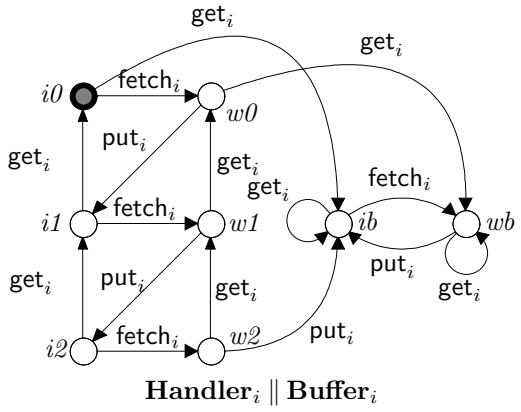


Figure 2. Synchronous composition of figure 1.

not including the termination action ω. In traditional supervisory control theory, the set of complete behaviours is described by a separate *marked language*; in this paper, the possibility of termination is indicated by the presence of ω in the language $L(P)$ of a process instead.

Note that for every prefix-closed language L there exists a deterministic process P such that $L = L(P)$. The construction of this process is straightforward. In the following, a prefix-closed language is identified with its accepting deterministic process and used in places where a process is expected.

When several processes are running in parallel, lockstep synchronisation in the style of [7] is used. The *synchronous composition* $P_1 \parallel P_2$ of two processes $P_1 = (\mathbf{A}_1, Q_1, \rightarrow_1, Q_1^\circ, Q_1^\omega)$ and $P_2 = (\mathbf{A}_2, Q_2, \rightarrow_2, Q_2^\circ, Q_2^\omega)$ is defined as

$$P_1 \parallel P_2 \stackrel{\text{def}}{=} (\mathbf{A}_1 \cup \mathbf{A}_2, Q_1 \times Q_2, \rightarrow, Q_1^\circ \times Q_2^\circ, Q_1^\omega \times Q_2^\omega), \tag{7}$$

where

- $(q_1, q_2) \stackrel{\alpha}{\rightarrow} (q_1', q_2')$ if $\alpha \in \mathbf{A}_1 \cap \mathbf{A}_2$ and $q_1 \stackrel{\alpha}{\rightarrow}_1 q_1'$ and $q_2 \stackrel{\alpha}{\rightarrow}_2 q_2'$;
- $(q_1, q) \stackrel{\alpha}{\rightarrow} (q_1', q)$ if $\alpha \in (\mathbf{A}_1 \setminus \mathbf{A}_2) \cup \{\tau\}$ and $q_1 \stackrel{\alpha}{\rightarrow}_1 q_1'$;
- $(q, q_2) \stackrel{\alpha}{\rightarrow} (q, q_2')$ if $\alpha \in (\mathbf{A}_2 \setminus \mathbf{A}_1) \cup \{\tau\}$ and $q_2 \stackrel{\alpha}{\rightarrow}_2 q_2'$.

The process in figure 2 is the synchronous composition of the two processes given in figure 1, but the state names have been shortened.

Furthermore, a process-algebraic *hiding* or *internalisation* operator is introduced in the standard way [16, 18]. Given a process $P \in \Pi_{\mathbf{A}}$, the result of hiding actions $\mathbf{A}' \subseteq \mathbf{A}$ is denoted by $P \setminus \mathbf{A}'$. The process $P \setminus \mathbf{A}' \in \Pi_{\mathbf{A} \setminus \mathbf{A}'}$ is constructed from P by replacing every occurrence of an action in $\mathbf{A}'$ by the hidden action τ.

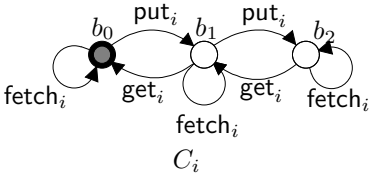


Figure 3. A candidate supervisor.

2.3 Controllability and Nonblocking

Controllability and nonblocking are concepts from supervisory control theory, presented in this section in a process algebraic setting. Controllability is motivated by the observation that control software can sometimes observe certain actions, like those associated to sensors, but cannot directly influence them. The nonblocking property makes sure that a system can always complete its task.

Supervisory control theory distinguishes between the system to be controlled, called the *plant* or *environment*, and the system that executes control, called the *supervisor* or *controller*. The supervisor interacts with the plant by disabling certain actions to achieve its control objectives.

However, the supervisor cannot control all actions. The alphabet of actions is partitioned into *controllable* actions and *uncontrollable* actions. A controllable action can be prevented from occurring by the supervisor, while an uncontrollable action cannot. The question arises whether a certain behaviour, given as a language, can be achieved by means of supervision. Such languages are called *controllable* [14].

Definition 2.1 Let $L, K \subseteq \mathbf{A}_\omega^*$ be two prefix-closed languages, and let $\mathbf{U} \subseteq \mathbf{A}$ be the set of uncontrollable actions. K is said to be $\mathbf{U}$-*controllable* with respect to L if $K\mathbf{U} \cap L \subseteq K$.

If a language K is controllable with respect to a plant behaviour L, then any uncontrollable action that is possible in the plant must also be possible in K. Clearly, K must not disallow an uncontrollable action to occur if it is physically possible in the plant, because no supervisor can stop uncontrollable actions from occurring.

Consider the language K given by the transition system in figure 3, which represents an attempt to build a supervisor to control the behaviour of the process in figure 2. K is not $\{\mathsf{put}_i\}$-controllable with respect to the language L given by figure 2 since $\mathsf{fetch}_i\mathsf{put}_i\mathsf{fetch}_i\mathsf{put}_i\mathsf{fetch}_i\mathsf{put}_i$ is contained in $K\mathbf{U}$ and L but not in K.

In addition to controllability, supervisors are required to perform some minimum functionality. In supervisory con-

trol theory, this is achieved by imposing a weak liveness condition, called *nonblocking*. It is required that the system is always able to complete its tasks, where completion of tasks can be represented using the set Q_ω of terminal states of a process, or equivalently by the termination action ω.

Definition 2.2 A process $P \in \Pi_{\mathbf{A}}$ is said to be *nonblocking* if for every trace $s \in \mathbf{A}^*$ and every $P' \in \Pi_{\mathbf{A}}$ such that $P \stackrel{s}{\Rightarrow} P'$ there exists a trace $t \in \mathbf{A}^*$ such that $P' \stackrel{t\omega}{\Rightarrow}$. Otherwise P is said to be *blocking*.

This definition is based on an implicit fairness assumption: the possibility of *divergence* is not considered as a problem. In order to be nonblocking, it is sufficient that every incomplete task *can* somehow be completed. As an example, process $\mathbf{Handler}_i$ in figure 1 is nonblocking, whereas process $\mathbf{Buffer}_i$ is blocking, because it is not possible to reach any terminal state from state *broken*.

3 Supervisory Control Synthesis

In traditional supervisory control theory, the plant is modelled as a generator of a language over an alphabet of actions, and the supervisor is a mapping from this language to the set of disabled or enabled actions. In an algebraic setting, plant and supervisor are considered to be processes. Given a plant $P \in \Pi_{\mathbf{A}}$ and a supervisor $C \in \Pi_{\mathbf{A}}$, the controlled behaviour of the plant under supervision is given by $P \parallel C$. The supervisor is assumed to be deterministic. The plant, and therefore also the controlled system $P \parallel C$, may be nondeterministic. This is a natural extension to the deterministic setting of traditional supervisory control theory.

The objective of supervisory control is to construct a supervisor for a given plant that meets certain *control objectives*. These control objectives are typically given as a *specification language*, representing the allowed behaviour of the system which must not be exceeded under any circumstances [15]. This paper adopts the viewpoint of [3] where it is shown that all control objectives can be expressed in terms of nonblocking. For example, a supervisor for plant $\mathbf{Buffer}_i$ in figure 1 has to prevent the process from entering state *broken*, because it would be blocking otherwise.

Given a plant $P \in \Pi_{\mathbf{A}}$, the supervisory control problem therefore is to find a supervisor $C \in \Pi_{\mathbf{A}}$ such that $P \parallel C$ is nonblocking. Furthermore, the supervisor C must not prevent uncontrollable actions from occurring, i.e., C must be controllable with respect to P. Thus, the supervisory control problem now consists of finding a supervisor within the following set of possible solutions.

Definition 3.1 Let $\mathbf{U} \subseteq \mathbf{A}$ be the set of uncontrollable actions. Define

$$\mathcal{C}_{\mathbf{U}}(P) \stackrel{\text{def}}{=} \{\, C \subseteq L(P) \mid C \text{ is } \mathbf{U}\text{-controllable with respect to } L(P) \text{ and } P \parallel C \text{ is nonblocking} \,\}. \quad (8)$$

Supervisory control theory provides means to compute a *behaviour* that can be achieved by a supervisor as a language $C \subseteq \mathbf{A}^*_\omega$. This language can be used to implement the actual supervisor as a process, simply by using a deterministic acceptor of the language.

An automatic synthesis algorithm must select one solution from $\mathcal{C}_{\mathbf{U}}(P)$. Supervisory control theory tries to identify a solution that restricts the plant as little as possible. But from the definition of $\mathcal{C}_{\mathbf{U}}(P)$ it is not immediately obvious whether such a least restrictive element exists. Therefore, it is now proven that the set $\mathcal{C}_{\mathbf{U}}(P)$ is closed under union of languages. Since this is already known for controllability, it suffices to consider the nonblocking condition. The following proposition extends the known result from [15] to the nondeterministic case considered here.

Proposition 3.1 Let $P \in \Pi_{\mathbf{A}}$ be a process, and let $(C_i)_{i\in I}$ be a family of languages over the alphabet $\mathbf{A}_\omega$. If $P \parallel C_i$ is nonblocking for all $i \in I$, then $P \parallel \bigcup_{i\in I} C_i$ is nonblocking.

Proof. Let $P \parallel C_i$ be nonblocking for each $i \in I$. Consider $s \in \mathbf{A}^*$ and $P', C' \in \Pi_{\mathbf{A}}$ such that $P \stackrel{s}{\Rightarrow} P'$ and $\bigcup_{i\in I} C_i \stackrel{s}{\Rightarrow} C'$. Since $\bigcup_{i\in I} C_i \stackrel{s}{\Rightarrow}$, it follows that $s \in \bigcup_{i\in I} C_i$. Therefore, there exists $k \in I$ such that $s \in C_k$. Since $P \parallel C_k$ is nonblocking, there exists t such that $P' \stackrel{t\omega}{\Rightarrow}$ and $st\omega \in C_k \subseteq \bigcup_{i\in I} C_i$. This proves that $P \parallel \bigcup_{i\in I} C_i$ is nonblocking. □

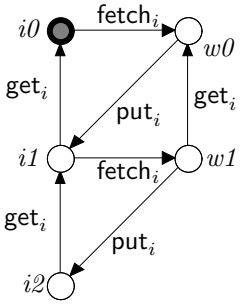
It follows that $\mathcal{C}_{\mathbf{U}}(P)$ is closed under union of languages. Therefore, this set contains a unique supremal element which can be used as a result of a supervisory synthesis algorithm.

Definition 3.2 Let $\mathbf{U} \subseteq \mathbf{A}$ and $P \in \Pi_{\mathbf{A}}$. Then the synthesis result for P with respect to $\mathbf{U}$ is given by

$$\sup \mathcal{C}_{\mathbf{U}}(P) \;=\; \bigcup \mathcal{C}_{\mathbf{U}}(P)\,. \quad (9)$$

The synthesis result for a given plant process P is a language, which can also be interpreted as a deterministic process, called supervisor. Running the plant and supervisor processes together yields a process whose behaviour is nonblocking. Furthermore, the supervisor never disables uncontrollable actions that are physically possible, and is least restrictive in the sense that it disables as few controllable actions as possible.

As an example, consider figure 1 once more. The two processes shown here model the plant behaviour of a small



$$S_i = \sup \mathcal{C}_{\mathbf{U}}(\mathbf{Handler}_i \parallel \mathbf{Buffer}_i)$$

Figure 4. Controller for the small factory.

factory consisting of a handler and a buffer. The handler fetches a workpiece (action fetch_i) and, when done, puts it into the buffer (action put_i), from where it can be collected (action get_i). The buffer itself cannot prevent overflow or underflow so a controller is required that ensures safe usage.

If a controller can prevent the physical device from putting a workpiece into the buffer, the process C_i in figure 3 can be used as a controller. Running the controller in parallel with the plant

$$\mathbf{Handler}_i \parallel \mathbf{Buffer}_i \parallel C_i \tag{10}$$

results in a nonblocking system where the buffer never overflows or underflows.

However, if the handler cannot be prevented from putting workpieces into the buffer (put_i is uncontrollable), the process C_i is not a feasible controller because it is trying to disable the uncontrollable action put_i when the buffer is full. Although the composition (10) remains nonblocking, the requirement of controllability rules out this behaviour as potential supervisor. In this case, supervisory control synthesis can be used to compute a least restrictive supervisor. The result is shown in figure 4.

This process can be automatically computed from the synchronous composition of the two plants, which is shown in figure 2. To ensure nonblocking, a would-be supervisor has to prevent the plant from entering states *ib* and *wb*. This can be achieved by disabling the controllable transitions labelled get_i originating from states *i0* and *w0*. However, the transition labelled put_i from state *w2* is uncontrollable and therefore cannot be disabled by a supervisor. Therefore, this state also must be considered as unsafe, because the plant can always execute the uncontrollable action put_i, and thus enter a blocking state. To ensure safe behaviour, a supervisor also has to prevent this state from being reached. This can be achieved by disabling the controllable transition labelled fetch_i in state *i2*. The resultant behaviour is controllable and nonblocking, and is shown in figure 4.

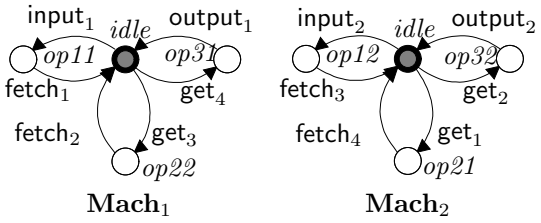


Figure 6. The machine processes.

4 A Modular Manufacturing System

The results presented so far can be used to synthesise controllers for relatively small examples only. The synthesis algorithm needs to compute the parallel composition of all the components to create the supervisor, which is not feasible for systems consisting of many components. Using the manufacturing system presented in [9], this section discusses how systems and controllers can be modelled and designed in a modular way. Modularity makes it possible to abstract away information that is not relevant in a particular context and thus makes it possible to handle huge systems.

Figure 5 gives an overview of the manufacturing system. It consists of two machines ($\mathbf{Mach}_1$ and $\mathbf{Mach}_2$) for processing workpieces and four subsystems ($\mathbf{Sub}_i$, $i = 1, \ldots, 4$) for moving and buffering workpieces in transit between the machines. Each subsystem consists of a buffer ($\mathbf{Buffer}_i$) that can store up to two workpieces, and a handler ($\mathbf{Handler}_i$) that fetches a workpiece from a machine and puts it into the buffer.

The manufacturing system can produce two types of workpieces. Type I workpieces are first processed by $\mathbf{Mach}_1$ (action input_1). Then they are passed through $\mathbf{Sub}_1$: they are fetched by $\mathbf{Handler}_1$ (fetch_1) and placed into $\mathbf{Buffer}_1$ (put_1). Next, they are processed by $\mathbf{Mach}_2$ (get_1), fetched by $\mathbf{Handler}_4$ (fetch_4) in $\mathbf{Sub}_4$ and placed into $\mathbf{Buffer}_4$ (put_4). Finally, they are processed by $\mathbf{Mach}_1$ once more (get_4), and released (output_1). Similarly, type II workpieces are first processed by $\mathbf{Mach}_2$, passed through $\mathbf{Sub}_3$, further processed by $\mathbf{Mach}_1$, passed through $\mathbf{Sub}_2$, and finally processed by $\mathbf{Mach}_2$. The behaviour of the machines is formalised in figure 6. The subsystems are modelled in figure 1; the index i should be instantiated with the number of the subsystem considered.

There are several requirements that the controlled system is expected to satisfy. Firstly, the buffers must never overflow nor underflow taking into account that actions put_i are uncontrollable. Another requirement is given as process $\mathbf{Toplevel}$ in figure 7, which is slightly stronger than in the original example [9]. It requires that the manufacturing system should produce type I and II workpieces in alternating sequence, starting with a type I workpiece. The objective to avoid blocking is also included—the initial state of the

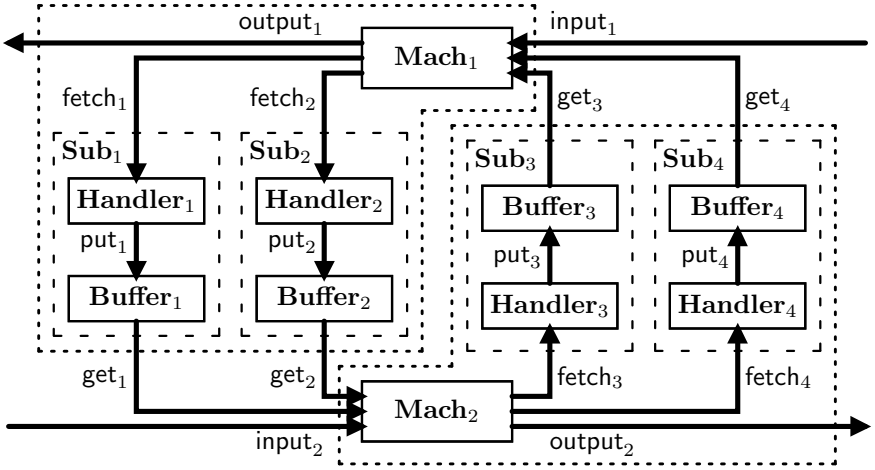


Figure 5. Manufacturing system example

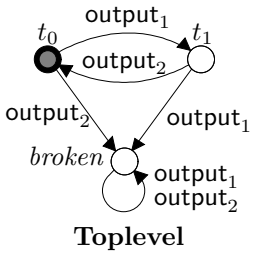


Figure 7. Top-level specification.

system must always remain reachable. This is not guaranteed in the original example: the solution given in [9] can deadlock by simply filling up all the buffers and machines.

The buffer requirements are ideally suited for a modular synthesis since they affect only a few components of the system. Only actions put_i and get_i need to be controlled to ensure that process $\mathbf{Buffer}_i$ does not reach state *broken*. Since the only uncontrollable actions, put_i, can only occur after fetch_i, it is sufficient to control action fetch_i to prevent undesired put_i actions. Thus, a controller for buffer i only needs to observe actions fetch_i, put_i, and get_i; it does not need to care about any other actions and how the rest of the system behaves. Therefore, a supervisor S_i for $\mathbf{Sub}_i$ can be computed from the plants concerned,

$$S_i \;=\; \sup \mathcal{C}_{\mathbf{U}}(\mathbf{Handler}_i \parallel \mathbf{Buffer}_i) \qquad (11)$$

where $\mathbf{U} = \{\mathsf{put}_i\}$. This construction has been discussed in section 3, and the result is shown in figure 4.

The subsystems

$$\mathbf{Handler}_i \parallel \mathbf{Buffer}_i \parallel S_i \qquad (12)$$

can now be considered black-boxes [18] for which only the externally observable behaviour is of interest. Figure 5 suggests that actions put_i can be considered as private communication between the handler, buffer, and local supervisor—they are not used anywhere else and thus do not need to be observable from the outside. These actions can be replaced by the hidden action τ. The black-box behaviour of the subsystems is thus given by

$$\mathbf{Sub}_i \;=\; (\mathbf{Handler}_i \parallel \mathbf{Buffer}_i \parallel S_i) \setminus \{\mathsf{put}_i\}\,. \qquad (13)$$

Hiding the internal actions in the subsystems alone does not bring much advantage since the size of the transition systems remains the same. However, often a transition system containing hidden actions captures more information than is necessary, and this information can be abstracted away.

5 Congruence Results

Abstraction and process equivalence are the key to modular modelling and analysis of large-scale systems. Given a system consisting of several components, the idea is to replace individual components by simpler *equivalent* ones, such that the crucial properties of the whole system are preserved.

There are different ways how two processes may be considered as equivalent. Traditional supervisory control theory uses deterministic processes and compares them according to their languages. This leads to a simple equivalence, known as *trace equivalence* in process algebra.

Definition 5.1 Two processes $P_1, P_2 \in \Pi_{\mathbf{A}}$ are said to be *trace equivalent* if their languages are equal, i.e., if $L(P_1) = L(P_2)$.

When considering nondeterministic behaviour, there are various other ways how processes can be considered as

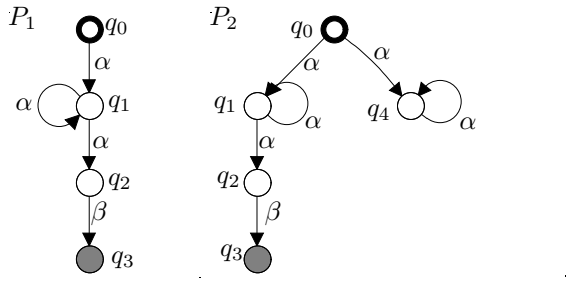


Figure 8. Two trace equivalent processes that do not have the same synthesis result.

equivalent, and other equivalence relations $\simeq \subseteq \Pi_{\mathbf{A}} \times \Pi_{\mathbf{A}}$ become of interest. Research in process algebra has identified, examined, and compared such relations—see [19] for a nice overview.

An important property of process equivalences is to be a *congruence* with respect to the operators used. Congruences guarantee that, if a component of a system is replaced by an equivalent one, the behaviour of the resulting system remains equivalent to the behaviour of the original system. This is the key property that makes modular reasoning possible.

Definition 5.2 Let $\simeq \subseteq \Pi_{\mathbf{A}} \times \Pi_{\mathbf{A}}$ be an equivalence relation.

- $\simeq$ is called a congruence with respect to synchronous composition, if $P_1 \simeq P_2$ implies $P_1 \parallel Q \simeq P_2 \parallel Q$ for every process $Q \in \Pi_{\mathbf{A}}$.
- $\simeq$ is called a congruence with respect to hiding if $P_1 \simeq P_2$ implies $P_1 \setminus \mathbf{A'} \simeq P_2 \setminus \mathbf{A'}$ for each $\mathbf{A'} \subseteq \mathbf{A}$.
- $\simeq$ is called a congruence with respect to synthesis, if $P_1 \simeq P_2$ implies $\sup \mathcal{C}_{\mathbf{U}}(P_1) \simeq \sup \mathcal{C}_{\mathbf{U}}(P_2)$.

The synthesis operator $\sup \mathcal{C}_{\mathbf{U}}$ can be used to compute new processes, i.e. supervisors, from existing ones, i.e. the processes to be controlled. For such an application of synthesis to be feasible, the underlying process equivalence has to be a congruence with respect to synthesis.

Not all equivalences satisfy this property. For example, figure 8 shows two trace equivalent processes P_1 and P_2, for which the synthesis results are quite different. Assuming all actions are controllable, the synthesis result for P_1 is $L(P_1)$, while the synthesis result for P_2 is empty. This proves that trace equivalence is not a congruence with respect to synthesis. The given example can also be used to show that traditional equivalences like ready-traces, failure traces, trajectories, and failures [19] are not a congruence with respect to synthesis.

It is of interest to see what kind of process equivalence would be a congruence with respect to synthesis. A closer inspection of the synthesis operator shows that it has only one aspect that poses problems as far as congruence is concerned—the requirement that the synthesis result be nonblocking. Indeed, this requirement means that the underlying equivalence must preserve conflicts in order to guarantee congruence.

Definition 5.3 An equivalence $\simeq$ on $\Pi_{\mathbf{A}}$ is said to *preserve conflicts* if $P_1 \simeq P_2$ implies that $P_1 \parallel T$ is blocking if and only if $P_2 \parallel T$ is blocking, for all $T \in \Pi_{\mathbf{A}}$.

Two processes that are equivalent with respect to a conflict-preserving equivalence behave equally with respect to blocking in combination with an arbitrary test process. If one of them is blocking with some test, so must the other be. The following result shows that any conflict-preserving equivalence is a congruence with respect to synthesis.

Theorem 5.1 *Let $\simeq$ be an equivalence on $\Pi_{\mathbf{A}}$ that preserves conflicts. Then, $P_1 \simeq P_2$ implies $\sup \mathcal{C}_{\mathbf{U}}(P_1) = \sup \mathcal{C}_{\mathbf{U}}(P_2)$.*

Proof. Let $\simeq$ be an equivalence on $\Pi_{\mathbf{A}}$ that preserves conflicts, and let $P_1 \simeq P_2$. It is sufficient to prove that $\sup \mathcal{C}_{\mathbf{U}}(P_1) \subseteq \sup \mathcal{C}_{\mathbf{U}}(P_2)$. Therefore, let $C = \sup \mathcal{C}_{\mathbf{U}}(P_1)$, and show that C is $\mathbf{U}$-controllable with respect to $L(P_2)$, and $P_2 \parallel C$ is nonblocking.

To see that C is $\mathbf{U}$-controllable with respect to $L(P_2)$, assume there is $s \in C$ and $v \in \mathbf{U}$ such that $sv \in L(P_2)$ and $sv \notin C$. Since C is the synthesis result for P_1, C is $\mathbf{U}$-controllable with respect to $L(P_1)$, i.e., $C\mathbf{U} \cap L(P_1) \subseteq C$. Therefore, $sv \notin L(P_1)$. Let T be a deterministic process with $L(T) = \{sv\} \cup C$. Then $T \stackrel{sv}{\Rightarrow} F$ where $F \in \Pi_{\mathbf{A}}$ is the process that refuses all actions. Since C is the synthesis result for P_1, $P_1 \parallel C$ is nonblocking. Because $sv \notin L(P_1)$, it follows by construction that $P_1 \parallel T$ is also nonblocking. But $P_2 \parallel T$ is blocking since $sv \in L(P_2)$ and therefore $P_2 \parallel T \stackrel{sv}{\Rightarrow} F$. This contradicts the assumption that $P_1 \simeq P_2$ and $\simeq$ preserves conflicts.

To see that $P_2 \parallel C$ is nonblocking, first note that $P_1 \parallel C$ is nonblocking. Since $P_1 \simeq P_2$ and because $\simeq$ preserves conflicts, $P_2 \parallel C$ is nonblocking. □

An equivalence used for modular composition of processes as suggested above should also be a congruence with respect to all the other operators used to model a system. This usually includes synchronous composition and hiding. If the equivalence already is a congruence with respect to synchronous composition, we can use the results from [10] to weaken the requirement of definition 5.3. In this case, it is sufficient to require that the equivalence preserves blocking.

Definition 5.4 An equivalence $\simeq$ on $\Pi_{\mathbf{A}}$ is said to *preserve blocking* if $P_1 \simeq P_2$ implies that P_1 is blocking if and only if P_2 is blocking.

Proposition 5.2 Let $\simeq$ be an equivalence on $\Pi_{\mathbf{A}}$ that is a congruence with respect to synchronous composition and preserves blocking. Then $\simeq$ preserves conflicts.

Proof. Let $P_1 \simeq P_2$, and let $T \in \Pi_{\mathbf{A}}$. Since $\simeq$ is a congruence with respect to synchronous composition, it holds that $P_1 \parallel T \simeq P_2 \parallel T$. Since $\simeq$ preserves blocking, it follows that $P_1 \parallel T$ is blocking if and only if $P_2 \parallel T$ is blocking. □

Thus, any process equivalence that is a congruence with respect to synchronous composition and respects blocking can be used for modular reasoning with the supervisor synthesis operator. Such equivalences exist. Indeed, definition 5.3 can be used directly to define an appropriate equivalence—*conflict equivalence*, the coarsest conflict-preserving equivalence [10]. Conflict equivalence is the coarsest equivalence that preserves blocking and is a congruence with respect to synchronous composition. Its main properties are repeated here for the sake of completeness.

Definition 5.5 (from [10]) Two processes $P_1, P_2 \in \Pi_{\mathbf{A}}$ are *conflict equivalent*, written $P_1 \simeq_{\text{conf}} P_2$, if it holds for every test $T \in \Pi_{\mathbf{A}}$ that $P_1 \parallel T$ is nonblocking if and only if $P_2 \parallel T$ is nonblocking.

Proposition 5.3 $\simeq_{\text{conf}}$ is a congruence with respect to $\parallel$ and respects blocking.

Proof (from [10]). First, let $P_1 \simeq_{\text{conf}} P_2$ and $T \in \Pi_{\mathbf{A}}$. To see that $P_1 \parallel T \simeq_{\text{conf}} P_2 \parallel T$, let $T' \in \Pi_{\mathbf{A}}$ be a test such that $(P_1 \parallel T) \parallel T'$ is nonblocking. Then $P_1 \parallel (T \parallel T') = (P_1 \parallel T) \parallel T'$ is nonblocking, and since $P_1 \simeq_{\text{conf}} P_2$, it follows that $(P_2 \parallel T) \parallel T' = P_2 \parallel (T \parallel T')$ is nonblocking. Therefore, $\simeq_{\text{conf}}$ is a congruence with respect to $\parallel$.

Second, note that there exists a process $U_{\mathbf{A}} \in \Pi_{\mathbf{A}}$ such that $P \parallel U_{\mathbf{A}} = P$ for every $P \in \Pi_{\mathbf{A}}$. Let $P_1 \simeq_{\text{conf}} P_2$, and let P_1 be nonblocking. Then $P_1 \parallel U_{\mathbf{A}} = P_1$ is nonblocking. Since $P_1 \simeq_{\text{conf}} P_2$, it follows that $P_2 = P_2 \parallel U_{\mathbf{A}}$ is nonblocking. Thus, $\simeq_{\text{conf}}$ respects blocking. □

Conflict equivalence furthermore can be shown to be a congruence with respect to hiding and other operators. Together with theorem 5.1, this implies that it is the coarsest equivalence that is a congruence with respect to synchronous composition, hiding, and synthesis, making it an ideal candidate for modular reasoning in supervisory control as suggested here.

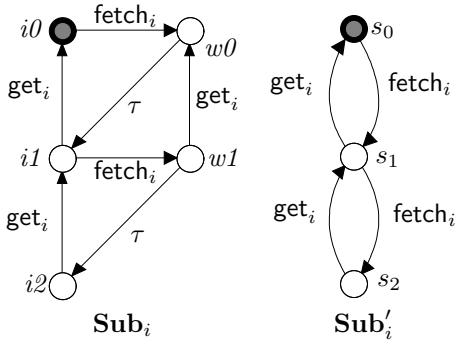


Figure 9. Simplifying subsystems.

6 Manufacturing System Continued

According to the congruence results from Section 5, it is possible to replace individual components of a large system designed using the operators parallel composition, hiding, and synthesis by conflict equivalent components. Doing so does not change, up to conflict equivalence, the behaviour of the result. This provides a powerful mechanism for the design of large controllers.

One possible application of the congruence results is the possibility to abstract away information that is not necessary. In the manufacturing example of section 4, it has been recognised that the subsystems consisting of a handler and a buffer can be controlled by local supervisors S_i. Then the uncontrollable actions put_i can be hidden from the rest of the system, resulting in

$$\mathbf{Sub}_i \;=\; (\mathbf{Handler}_i \parallel \mathbf{Buffer}_i \parallel S_i) \setminus \{\mathsf{put}_i\}\,. \quad (14)$$

The five-state processes $\mathbf{Sub}_i$ are conflict equivalent to the three-state processes $\mathbf{Sub}'_i$ as shown in figure 9—in fact, bisimulations can be used to verify this [10, 11]. Using transition systems with fewer states and transitions makes the computation of subsequent operations cheaper, and for large systems possible at all.

This modular construction of supervisors can be continued. In a next step, it can be noted that actions fetch_1 and fetch_2 are needed only for communication between machine $\mathbf{Mach}_1$ and subsystems $\mathbf{Sub}_1$ and $\mathbf{Sub}_2$. Therefore, it is worth considering the new subsystem

$$M_1 = (\mathbf{Mach}_1 \parallel \mathbf{Sub}'_1 \parallel \mathbf{Sub}'_2) \setminus \{\mathsf{fetch}_1, \mathsf{fetch}_2\} \quad (15)$$

as a black-box. This process M_1 has 36 states and turns out to be nonblocking, so no synthesis step is needed. Using simplification procedures from [1, 2], this process can be replaced by a 27-state conflict equivalent process M'_1, which yields an equivalent behaviour of the overall system as the larger process M_1. Likewise, a simplified subsystem M'_2 can be constructed from $\mathbf{Mach}_2$, $\mathbf{Sub}_3$, and $\mathbf{Sub}_4$ by hiding actions fetch_3 and fetch_4.

Finally, the simplified subsystem models are used to synthesise a controller for the entire system,

$$S \;=\; \sup \mathcal{C}_{\mathbf{U}}(M_1' \parallel M_2' \parallel \mathbf{Toplevel}) \;. \qquad (16)$$

The synthesised supervisor S has 1,356 states. In combination with the local supervisors for the four subsystems, this yields a modular supervisor for the entire manufacturing system. By the results of section 5, the modular supervisor is controllable and nonblocking. By theorem 5.1, it achieves the same behaviour as any least restrictive monolithic supervisor that does not make use of the hidden actions $\mathtt{put}_i$ and $\mathtt{fetch}_i$.

The manufacturing example in section 5 is small enough to allow the computation of a monolithic supervisor that satisfies all requirements, is least restrictive, and guarantees controllability and nonblocking using the standard algorithms from [15]. The supervisor obtained in this way has 17,038 states, which is much bigger than the modular supervisors.

A great achievement of a modular design is that subcomponents can be replaced by processes that are conflict equivalent without affecting the overall behaviour of the system, and without having to synthesise supervisors again that use those subcomponents.

This sort of substitutability is not possible in a centralised design. A centralised supervisor does not have a black-box view of subcomponents but can observe all actions. While this generally leads to much more complicated supervisors, it allows the synthesis of a *least restrictive* supervisor that achieves the control objective in the most optimal way. In general, hiding actions and thus making them invisible for controllers constructed at later stages may force those controllers to prevent behaviour that would otherwise be safe.

In the example, the hiding of the $\mathtt{put}_i$ and $\mathtt{fetch}_i$ actions have been deliberate design decisions, based on the structure of the overall system. In this particular case, the resultant supervisor turns out to yield a least restrictive behaviour. In general, it is up to the designer to decide under which circumstances least restrictiveness or modularity is more important.

7 Related Work

Similar attempts to extend supervisory control theory have been made by several researchers, in order to introduce nondeterminism or to make the large set of process-algebraic operators available and achieve modularity. However, most researchers in supervisory control theory have paid little attention to congruence results.

Heymann [4] proposes a framework for supervisory control based on the failure trace semantics, which has been extended to handle nonblocking [5]. He only considers a weak kind of congruence, called *language-congruence*. Language-congruence ensures that applying an operator to equivalent processes results in processes that have the same language, but are not necessarily equivalent. Congruence results with respect to the synthesis operator are not considered.

Shayman and Kumar [17] provide another framework based on the trajectory model, which has congruence results. The framework has also been extended to handle nonblocking [8]. This approach is closely linked to the trajectory model, which requires the use of unusual concepts of hiding and nonblocking.

Overkamp [12, 13] introduces supervisory control of nondeterministic systems based on failures semantics [7]. In this framework, only deadlock-freedom is considered, and a pessimistic approach toward divergence is taken. This approach therefore differs significantly from the original supervisory control theory.

Recently, researchers in supervisory control theory have made some efforts to achieve modular synthesis. Hill and Tilbury [6] use *language projection* to simplify intermediate results. Their approach is limited to deterministic automata and the restricted abstraction potential of projection. In [3], an alternative approach using *state labelling* is discussed which, while providing encouraging results, is much more complicated and also fails to have the full abstraction potential of process equivalence.

This paper proposes a process-algebraic framework that is closely linked to the original supervisory control theory, using concepts of synthesis and nonblocking that are immediate extensions of the traditional concepts to the case of nondeterminism. It studies the process equivalence needed to ensure congruence results for all operations including synthesis.

8 Conclusions

The objective of this research is to compare process algebra and supervisory control theory and combine the advantages of both. Process algebra provides modularity, while supervisory control provides synthesis algorithms. Even though the two research areas have many concepts and goals in common, there are subtle differences that make communication difficult.

As a first step towards a combination of these two fields, this paper describes supervisory control theory using process-algebraic terminology. The original supervisory control synthesis based on formal languages is extended in a natural way to handle nondeterministic processes. This makes it possible to examine congruence results with respect to synthesis. The main result of this paper is that it points out the close connection between such congruence

results and nonblocking: in order to obtain congruence with respect to synthesis, and thus in order to guarantee modularity, it suffices to use a conflict-preserving equivalence. The example presented in section 6 demonstrates how such an equivalence can be applied to address the long-standing problem of synthesising modular nonblocking supervisors.

This work can be extended in several ways. The authors have started to work on algorithms to minimise finite-state systems in such a way that conflict equivalence is preserved. A present solution uses heuristics and yields good results in many cases [1,2]. Work is in progress to extend this and implement effective decision and minimisation procedures for conflict equivalence. In the future, the authors would like to extend the framework to consider least restrictive supervisors, build a tool to design large systems of processes using synthesis in a modular way, and apply it to more realistic applications.

References

[1] H. Flordal. *Compositional Approaches in Supervisory Control*. PhD thesis, Chalmers University of Technology, Göteborg, Sweden, 2006.

[2] H. Flordal and R. Malik. Modular nonblocking verification using conflict equivalence. In *Proc. 8th Int. Workshop on Discrete Event Systems, WODES'06*, pages 100–106, Ann Arbor, MI, USA, July 2006.

[3] H. Flordal and R. Malik. Supervision equivalence. In *Proc. 8th Int. Workshop on Discrete Event Systems, WODES'06*, pages 155–160, Ann Arbor, MI, USA, July 2006.

[4] M. Heymann. Concurrency and discrete event control. *IEEE Control Syst. Mag.*, June 1990.

[5] M. Heymann and F. Lin. Nonblocking supervisory control of nondeterministic systems. Technical Report CIS Report 9620, Technion – Israel Inst. of Technology, Haifa, Israel, 1996.

[6] R. C. Hill and D. M. Tilbury. Modular supervisory control of discrete-event systems with abstractions and incremental hierarchical construction. In *Proc. 8th Int. Workshop on Discrete Event Systems, WODES'06*, pages 399–406, Ann Arbor, MI, USA, July 2006.

[7] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[8] R. Kumar and M. A. Shayman. Non-blocking supervisory control of nondeterministic systems via prioritized synchronization. *IEEE Trans. Automat. Contr.*, 41(8):1160–1175, Aug. 1996.

[9] F. Lin and W. M. Wonham. Decentralized control and coordination of discrete-event systems with partial observation. *IEEE Trans. Automat. Contr.*, 35(12):1330–1337, Dec. 1990.

[10] R. Malik, D. Streader, and S. Reeves. Conflicts and fair testing. *Int. J. Foundations of Computer Science*, 17(4):797–813, 2006.

[11] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer, 1980.

[12] A. Overkamp. Supervisory control for nondeterministic systems. Technical Report BS-R9411, Dept. of Operations Research, Statistics, and System Theory, CWI, Amsterdam, The Netherlands, 1994.

[13] A. Overkamp. Supervisory control using failure semantics and partial specifications. *IEEE Trans. Automat. Contr.*, 42(4), Apr. 1997.

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

[15] P. J. G. Ramadge and W. M. Wonham. The control of discrete event systems. *Proc. IEEE*, 77(1):81–98, Jan. 1989.

[16] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.

[17] M. A. Shayman and R. Kumar. Supervisory control of nondeterministic systems with driven events via prioritized synchronization and trajectory models. *SIAM J. Control and Optimization*, 33(2):469–497, 1995.

[18] A. Valmari. Compositionality in state space verification methods. In *Proc. 18th Int. Conf. Application and Theory of Petri Nets*, volume 1091 of *LNCS*, pages 29–56, Osaka, Japan, June 1996. Springer.

[19] R. J. van Glabbeek. The linear time —branching time spectrum I: The semantics of concrete, sequential processes. In J. A. Bergstra, A. Ponse, and S. A. Smolka, editors, *Handbook of Process Algebra*, pages 3–99. Elsevier, 2001.